1982
APL Users Meeting

Proceedings Volume II

Special Technical Topics
Designing APL Systems

Sponsored by:
I.P. Sharp Associates Limited

APL Users Meeting Proceedings Volume II 1982

1982
APL Users Meeting

Proceedings Volume II

Special Technical Topics
Designing APL Systems

I.P. Sharp Associates Limited
Toronto, Ontario
1982

PREFACE

The idea for this publication arose during the planning meetings for the 1982 APL Users Meeting. The planners decided to have a series of special technical topics at the conference which would collect and organize the key concepts in the design of APL systems. These papers are the result.

The advice we offer in this book reflects the ideas of many, many people. Particular contributions were made by Peter Airs, Walter Fil, Leslie Goldsmith, Ken Iverson, Doug Keenan, Clement Kent, Jane Minett, and Henri Schueler.

The artwork and layout were done by Susan Storey. Text-editing was done by Maureen Groom, Margot Murray, and Deborah Rodbourne, while David Manson shepherded the papers through the many stages of production. The volume would never have reached publication without Rosanne Wild's unfailing support and sound advice.

We hope you learn as much from reading these papers as we did from writing them.

L Gibson
J. Levine
R. Metzger

Toronto, 1982

Printed in Canada
Pub Code: 0184 8210 E1
ISBN 0-86493-071-2

TABLE OF CONTENTS

Original artwork by Susan S. Storey

DESIGNING USER-FRIENDLY APL SYSTEMS

Lib Gibson
Manager, Applications Software
I.P. Sharp Associates Limited
Toronto, Ontario

INTRODUCTION

The expression *user-friendly* is a buzzword for both the computer community and the general public. "Friendly systems" are deemed to be an inalienable right of every computer user, and the delivery of such systems the unshirkable duty of every systems analyst. It is a charter of rights easier to articulate than to live up to.

User-friendliness is considered to be an important right because human resources are becoming more and more expensive. Anything that can improve productivity is important—and friendly computer systems raise productivity:

- Users can learn to use them more quickly and require less initial training.
- Users complete their tasks more quickly and with fewer errors.
- Less support is required during the life of the system.
- Users are more self-sufficient, able to carry out more ad hoc analyses without the involvement of a professional programmer.
- In many systems, the people who do the data input are not the ones who use the data. A friendly system can make these data entry people more accurate and timely by improving their general attitude to the system.

The user's interaction with your system and whether that interaction is considered "friendly" is the first aspect of a system the user meets, and is the first criterion on which a system is judged. This is the part of the system that you are stuck with right through to the (perhaps bitter) end; users resent changes—even "improvements"—in the behaviour of a system once they have grown accustomed to it. For instance, APL language implementors have rewritten parts of the interpreter to improve performance or maintainability, but the user interface, the syntax of the APL language and the definition of primitive functions and operators, has remained stable.

Once you have released an application system, you'll have unlimited freedom to *extend* it—in fact, if you achieve any sort of success, you'll be besieged with requests for extensions. However, because many users will already have been trained, and because other applications will have been built around your syntax, you'll be locked

into the system's general style and syntax. You will be constrained from *changing* the existing features, except in an upwardly compatible way. You've got to get it right from the start. The success or failure of your system depends on it; friendliness is a necessary (but not necessarily a sufficient) condition for success.

This paper surveys approaches and techniques for designing user-friendly systems in APL, starting with a fundamental design and implementation philosophy. Two major styles of systems, open and closed, are described and contrasted, and a method to choose between the two is presented. Some do's and don't's are given for language design in each type of system. The important topic of robustness is treated next, and input validation, restartability, bug files, and reliability are covered. The section on hardware considerations shows how to take advantage of terminal features, including graphics and full screen devices. The final section covers further user considerations such as user profiles, the provision of documentation, and outward compatibility.

In the course of these discussions, I will make many recommendations for designing user-friendly systems. These are based on experience as a user of APL systems, and as a referee (or official devil's advocate) of systems before release. Based on these experiences, I will also show you examples of things *not* to do. You may consider some of these examples glaringly obvious. But I can assure you that I have seen examples of all the blunders I discuss; I've just changed the (function) names to protect the guilty. Just as naughty children go sometimes undisciplined by doting parents, so a program's behavioural aberrations can be fondly overlooked or forgiven by its indulgent programmer. If you can persuade someone to check your system for such blunders, be it a user or an uninvolved bystander, do so. If not, use a checklist such as the one which appears in the Appendix to ruthlessly examine your own systems and discipline your own program's behavioural problems.

I must include a cautionary note here—there is a cost associated with designing a truly user-friendly system:

- Such a friendly system can be more expensive to run from the point of view of machine resources. Many features which make a system more friendly take their toll in CPU units. Be careful, though, not to overestimate the computer costs: the benefits of friendliness can be difficult to measure. Friendliness can mean the user achieves his task more quickly, has fewer false starts and errors, and so actually uses the program fewer times. But this sort of shift in usage pattern is difficult to measure in a benchmark situation which merely compares one clean run with another clean run.

- There is development cost to consider. A good rule of thumb is that it takes at least ten times as much effort to write a generalized, user-friendly system, usable by a wide range of users, as to solve the same problem in a particular way, for purely personal use.

There are always trade-offs to be made. As a systems designer you must weigh the advantages of user-friendliness versus maintainability and versus efficiency. One thing is clear, however: doing it right and making it friendly from the start will increase the chances of your system being successful and being used effectively. It may also save on maintenance later.

MAKING FRIENDS DURING THE DESIGN PROCESS

How you design your system is almost as important as *what* you design in establishing friendly relations with users. As designers of APL applications, you are fortunate to be using a tool that facilitates flexible solutions, which can be responsive to users' needs. Indeed, the very fact that you are working with an interactive timesharing system is a great advantage.

When designing a specialized application system, you should use an evolutionary design process. This approach is particularly well suited to APL, with its low implementation costs. In an evolutionary approach, the designer/programmer (you inevitably play both roles) builds a preliminary, or prototype system, or bits and pieces of a system in response to the user's initial (usually ill-defined) specifications. This prototype system then serves to draw out more clearly the user's needs, and is really part of the specification process—not the design process, and certainly not the implementation process. This first system will probably be thrown away. In fact, you'd prefer to throw it away. You don't want to have to maintain upward compatibility (as discussed later) with a system that embodies misconceptions (yours and the user's) and misunderstandings.

You then build the real system. This may again be modified according to the user's response. Several more of these iterations can take place, and the system eventually evolves to address the problem very, very well. If the system is successful and remains in use, this cycle will never end, although it will slow down. The problem will keep changing, the users will become more sophisticated, new capabilities will be added to APL—and the solution will have to adapt. Continuing an evolutionary process in the "mature growth" period of your system maximizes your chances of successfully addressing this challenge of adaptation. It ensures that you are in frequent and meaningful contact with the user and the problem, and that you have direct (and we hope nonviolent) feedback. Writing a maintainable system allows you to keep your sanity during this whole procedure.

Designing a general system, such as a data base management system, or a statistical forecasting system, is clearly a more difficult task, because, by definition, it takes place one step removed from an individual problem and user. As you design the system, you don't clearly know who the users will be. Such a system can be developed with one user standing in for the whole intended population; you can then proceed as for a specialized system. But the risk is great that the system will reflect too narrowly one individual's problems and tastes. The only solution is to make sure the general system goes through the same evolutionary growth as does the special purpose system.

In both the general and the specific cases, it is vital to start simply. You must resist the temptation to incorporate every possible feature and capability, or the result will be a system that is too complex and too expensive. When you have to issue the manual with casters attached, and the local university starts granting postgraduate degrees in your system, you'll know you've gone too far. So be cautious. *Plan* for all the marvellous features and capabilities in the design but don't implement them until user feedback demands them. If no one asks for them, they probably weren't worthwhile features anyway. Use this simple rule: assess whether 90% of the users will use the new feature; if not, you probably shouldn't implement it.

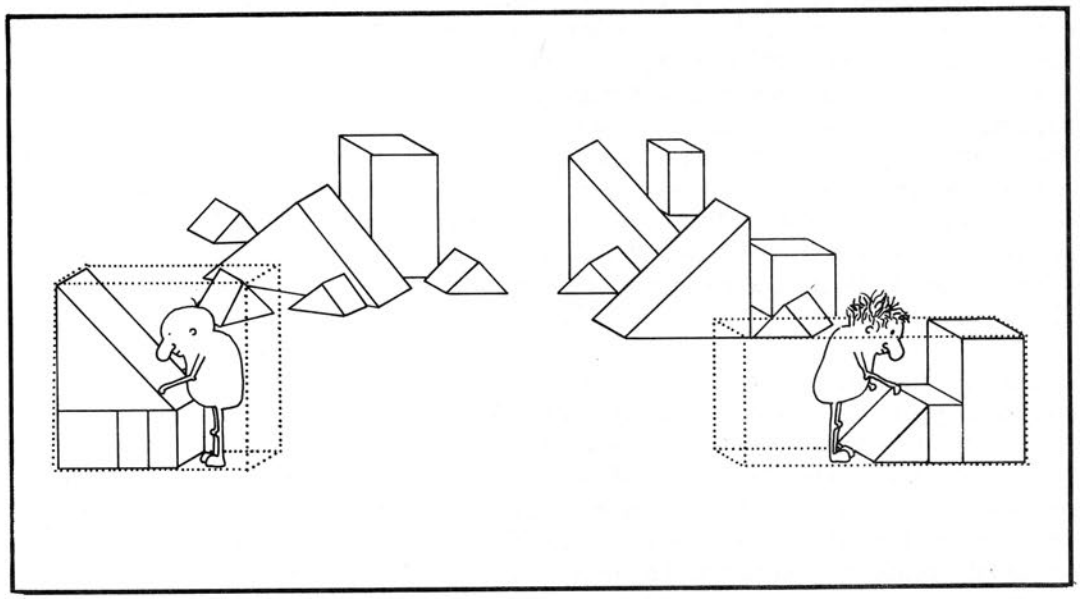
CHOOSING THE STYLE OF THE SYSTEM

A critical decision in system design is the choice of style for the system. There are two radically different styles of systems from which to choose when embarking on an APL system design: open and closed.

An **open system** is essentially a set of functions which are designed to fit together conveniently to perform certain types of tasks. The user spends his time in immediate-execution mode, with a number of functions available for use. Since the user works directly in an APL workspace, the grammar, or syntax, of the application is that of APL itself. The application merely adds more vocabulary, in the form of verbs (functions) and nouns (variables) to APL (whose vocabulary consists of the primitive functions). No prompting is involved and the user is free to put together the functions in whatever manner he chooses; after the execution of each function, he is returned to immediate-execution mode.

An open system can be pictured as in Figure 1.

Figure 1



The user picks up the modules (functions) of his choice and builds them into the system he has in mind. Different users may select different functions, and fashion them into rather different end systems.

A collection of utility functions satisfies these criteria, as long as you add the "glue" of personal APL functions. But an open system goes a step further, in that the functions themselves form a complete system, which can perform the given task without any programming by the user. So although he is in immediate-execution mode, the user does not *need* to know any APL. The functions in an open system can be strung together to form phrases:

```
      DISPLAY SALES AND EXPENSES, ONLY 1 80 TO 12 82

      SORT COUNTRY BESIDE CAPITAL BESIDE RANK POPULATION
```

An open system is embedded directly in an APL workspace, and its commands are interpreted directly by the APL syntax analyzer. As a result, APL errors are freely shown to the user. For instance, if a function name is mistyped, an untreated APL message, *VALUE ERROR*, is produced:

```
      DISPLAT SALES AND EXPENSES
VALUE ERROR
      DISPLAT SALES AND EXPENSES
      ^
```

A user with a bit of APL knowledge can create very powerful, customized functions by using an application's functions as "high-level primitives". He is in effect writing prose with the application system vocabulary and APL's grammar rules. These customized functions can extend the applicability of the system beyond even the expectations of the designer. The only limit is the richness of the basic tools and the imagination and creativity of the user. While the user is building these functions, he is learning about APL grammar, and also about the mechanics of writing and editing an APL function.

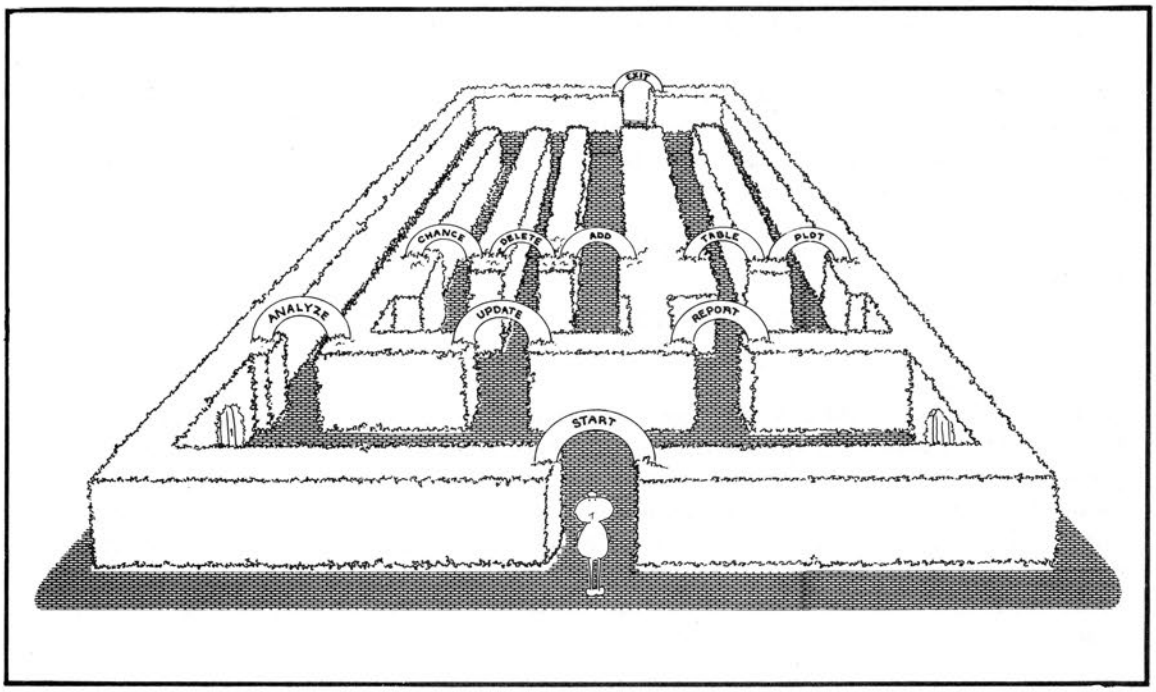
With a little more knowledge of APL primitive functions, the user can intermingle raw APL with the application's functions as he further customizes. Since the system's functions were designed to behave like APL primitives, they fit easily and conveniently into APL itself.

And finally, the user can go even further and write new primitive functions for the application system itself. In this way, he can extend the basic vocabulary of the system.

The beauty of an open system is that it is so easy to extend this way. The user does not have to understand the internal workings of the system to be able to extend it. He builds modules quite independent of the existing ones, and needs only to consider the existing external behaviour of the system.

A **closed system** usually has only *one* command that is in immediate-execution mode. From that point on, the program is, essentially, in control of the user, rather than vice versa. The application designer is faced with constructing a new set of grammar rules, since he can no longer rely on those of APL. The process can be pictured as a user finding his way through a maze (Figure 2). His vision at any point is limited to a certain set of options. He cannot choose (or even see) options possible down other paths. Once he has chosen the activity he wants to perform, he must complete all the steps on the selected path before he reaches the point where he can choose some other task.

Figure 2



The user is guided through this maze by the dialogue of a closed system. A prompt is issued to the user, and one of a limited set of legal responses is accepted, which determines what the next prompt will be [Note 1].

```
      COMMAND: CHANGE
PARAMETER/CASE: PARAMETER
```

If the user's response is invalid in any way, the program issues an error report, and reprompts:

```
      COMMAND: CHANGE
PARAMETER/CASE: VARIABLE

**ERROR- ENTER ONE OF PARAMETER OR CASE
PARAMETER/CASE:
```

The user never sees error messages from APL itself; they are always generated by the application system, in the terms of the application itself.

The system provides information and guidance on how to use it in response to a user's request.

In combination, these features of a closed system can combine to create an environment with a high "comfort level" for naive users. In particular, it is extremely easy to get started using a closed system.

How can you decide whether an open or closed system is more appropriate to your problem? Well, you can't decide which is the "better" vehicle, a tank or a race car, until you find out the purpose. So we can't decide about application systems without an appraisal of the purpose. We want to be *user*-friendly, so let's start with our users.

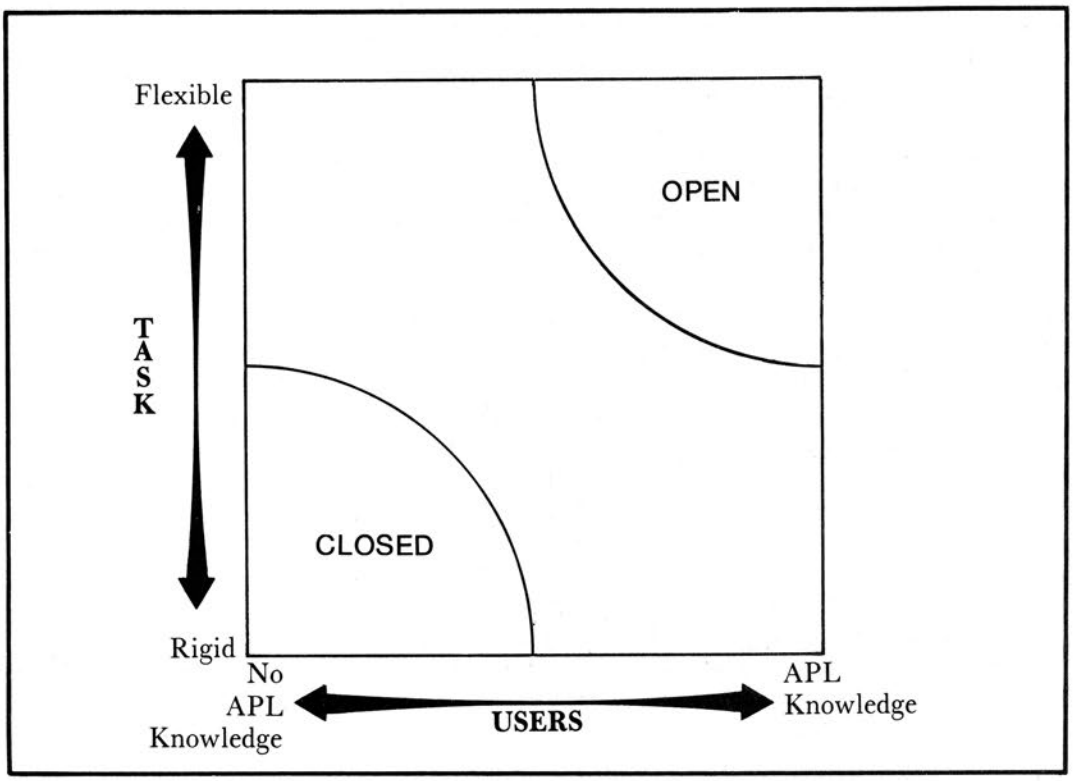
Any time you are writing something for someone else—be it a letter, a book, or a program—it is important to keep the audience for whom you are writing uppermost in your mind. Our audience is the *user* in user-friendly. Different sorts of users want different sorts of friends. Those who use computers to assist them in decision making and high level planning aspire to having at their fingertips a willing and able slave; those involved in routine administrative tasks at best hope for a benign dictator. Most commonly, these users are not computer professionals. Such naive users, naive at least about computers, prefer the comfort and protection a closed system provides.

Another user is often neglected: the knowledgeable programmer (not necessarily the original author of the package) who must personally use a system, or adapt an existing system for others. Such a user demands a friend who is a true collaborative colleague, to interact with and to sometimes stimulate with new ideas. He prefers the freedom to exploit his knowledge in an open system.

We must also consider the task being performed. Each type of system enjoys an advantage in a different environment. Closed systems are most effective in a highly structured, rigid environment, performing repetitive, unvarying tasks. Good examples are data entry systems and narrow inquiry systems. The more flexible or unstructured the environment, the more appropriate an open system. Such a system may take slightly longer to learn, but offers greater scope for exploitation, adaptation, extension, and enhancement than the rigidly structured closed system. Good examples are decision support systems, or any system where the unanticipated predominates.

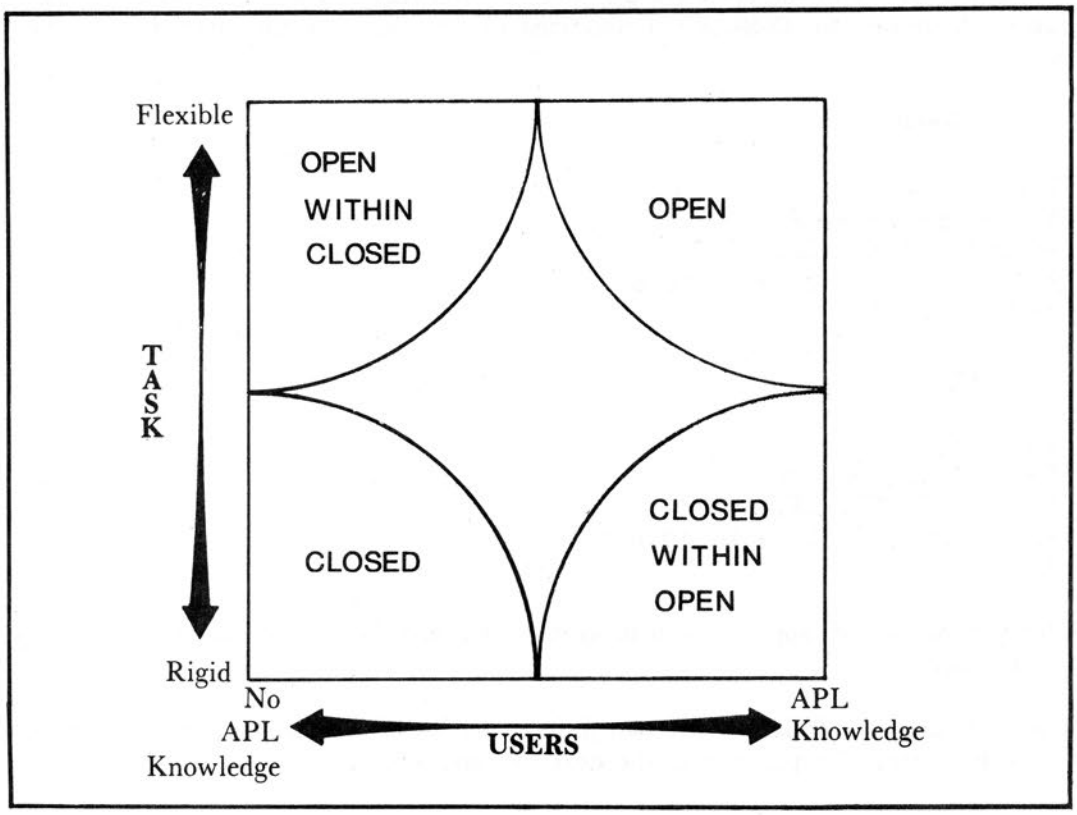
We can chart the areas where the different systems excel as in Figure 3.

Figure 3



This leaves two situations not covered—the repetitive task being done by an APLer, and the highly unstructured task being done by the non-APLer. The former is best handled by embedding closed functions in an open system. For instance, you can build a closed function for the data entry part of the system, but still let the user "live" in a free APL workspace for the analytic part. For the latter case, the best approach is to embed open functions within a controlling interface (closed system) which just interprets errors, offers help, and generally provides a somewhat more comforting environment. This type of system is typified by the case where a closed system allows the user to enter a rather free form model, APL-like in syntax perhaps, but provides a buffer between the user and the raw APL environment, which customizes error messages and offers help. So we can finish off our diagram, as in Figure 4.

Figure 4



This still leaves an area where it is not clear which style of system is best. Besides, most systems need to satisfy both types of tasks, and a mix of users; thus the best approach is to build both an open and a closed system. Then each user can choose which version to use, depending on his own taste and the task at hand.

An open system can be easily "enclosed" but it may be very difficult to "open up" a closed system. Thus, even though the impetus for the system may be coming from a user who wants and needs a completely closed system, it is important to start with an open system. You can use the functions of the open system in building the closed system. In the process, you will discover whether these functions represent a convenient, usable tool.

Although the programmer starts off with the open system, the naive user starts off with the closed system. Once he has gained confidence with the closed system, he may want to give up the comfort of this enclosed space and head for the wide open spaces of an APL workspace. Now that he has gained familiarity with the closed system, he can exploit the freedom of the open version.

In many systems, it is possible to give such a user a head start on the use of the companion open system by planning that the prompts in the closed system correspond to function names in the open system, and that the required responses are the arguments. (A prompt for *COMMAND* is converted to a straightforward function call.) For example:

Closed

```
FILE: SALES
REGION: EUR,USA,JAP
PRODUCT: LAGER,ALE,STOUT
PERIOD: MONTHS, 1 79 TO 10 82
COMMAND: LIST
```

Open

```
FILE 'SALES'
REGION 'EUR,USA,JAP'
PRODUCT 'LAGER,ALE,STOUT'
PERIOD 'MONTHS, 1 79 TO 10 82'
LIST
```

This sort of parallel approach can make the transition from closed to open very easy for the user.

The approaches to designing the two different styles of systems are rather different, and will be treated separately in the next two sections.

LANGUAGE DESIGN IN OPEN SYSTEMS

When you design any application system, you are in effect inventing a language, a creative exercise that is part art and part science. This is never more apparent than when designing an open system. Since you are designing a language, you have an excellent role model to emulate—APL itself. The very attributes we admire in APL are the ones you should strive for in the systems you write. Let us list some features that apply to the design of an open system:

- flexibility and generality
- applicability to arrays
- use of state settings
- consistency

Flexibility and Generality

APL's flexibility arises from its syntactic design—functions have one or two arguments and explicit results. The result of one function can form the argument of the next one, to build up ever more complex expressions. An open application system should be built in the same style: a number of user-defined functions with arguments and results, where the results of one function naturally form the arguments of another. The user is free to put together these functions in a variety of ways. Judicious choice of function names enables you to create a language in which English statements may be constructed:

```
COMPARE BUDGET WITH ACTUAL

PLOT TREND REVENUE

READ 'SALESHISTORY' FROM FILE 'DEPRESSING'

STORE 'SALESFORECAST' IN FILE 'OPTIMISTIC'
```

By providing the building blocks and designing them with explicit results, such phrases can be constructed in a meaningful way with "connective" functions such as *IN*, *FROM*, and *WITH* to make the language flow more easily. (A powerful side benefit is that in a system such as the above, should a complete file redesign be needed, only two functions, *READ* and *STORE*, need to be modified.)

Let us see how some sample open systems give the user the freedom to do the three kinds of customization previously described, namely:

- building APL functions using only the functions of the system
- mixing in raw APL
- creating new primitive functions for the system

Consider a system which offers network analysis and project management. One could write a function called *CRITICALREPORT* which provides a standard report on the critical activities in the system:

```
      CRITICALREPORT
PRED    SUC     DESC
101     102     RECEIVE TENDERS
102     103     AWARD CONTRACT
 .       .            .
 .       .            .
 .       .            .
```

However, your users are much better served by two functions: *CRITICAL*, which generates the list of critical activities, and *DISPLAY*, which produces the above report by typing:

```
      DISPLAY CRITICAL
```

Then, you can go further and allow a left argument of *DISPLAY* to specify what fields or columns you want to appear in the report. In the example below, the user asks for **E**arly **S**tart and **E**arly **F**inish as well:

```
      'PRED,SUC,DESC,ES,EF' DISPLAY CRITICAL

PRED     SUC     DESC              ES      EF
101      102     RECEIVE TENDERS    0       4
102      103     AWARD CONTRACT     4       6
 .        .           .            .       .
 .        .           .            .       .
 .        .           .            .       .
```

In fact, you can offer the user even more convenience by adding functions *SORTED*, *BY*, *NOT*, and *WITHIN* which can be used together as follows:

```
      'DESC,DEPT,DUR,TF' DISPLAY ALL SORTED BY 'TF'
      DISPLAY (NOT DUMMIES) SORTED BY 'ES' WITHIN 'DEPT'
```

Flexibility and generality in part arise from modularity. The key is to provide functions at a low enough level to allow the user to "mix and match". For instance, you could be tempted to have the reports printed by *DISPLAY* always sorted on, say, the first column. Separating out the *SORTED* function gives the user absolute control over how the sorting is done, and results in much greater flexibilty and power.

Suppose the manager of a project signs on every day to run off a standard report to see which activities must be initiated today (those whose **S**cheduled **S**tart is today), or are scheduled to finish today (those whose **S**cheduled **F**inish is today). He can easily define this task as an APL function to save typing:

```
    ∇ DAILY
[1]   NETWORK 'BIG PROJECT'
[2]   'ACTIVITIES TO BE STARTED ',TODAY
[3]   'DESC,DEPT,SF' DISPLAY SS EQ TODAY
[4]   'ACTIVITIES DUE FOR COMPLETION ',TODAY
[5]   'DESC,DEPT' DISPLAY SF EQ TODAY
      ∇
```

The user has used functions such as *NETWORK*, *TODAY*, *EQ* to build his own APL program without knowing APL.

Modularization is again evident in the next example. This example shows that users can integrate their APL knowledge easily into an open system. Let us suppose an electronic mailbox offers a function to *SUMMARIZE* all *INCOMING* messages:

```
SUMMARIZE INCOMING
1847654      PCB
1862171      KEI
1868243      IPS
```

A user who wants to *DISPLAY* only those messages from someone with a particular mailbox code, can write a function combining the functions of the application and APL:

```
    ∇ SHOW NAME;M
[1]   ⍝ DISPLAY ALL MESSAGES FROM <NAME>
[2]   M←SUMMARIZE INCOMING
[3]   DISPLAY ⎕FI,((0 19↓M)∧.=5↑NAME)⌿M[;+⍳19]
    ∇
```

If he does this kind of selection regularly enough, it is simple to write a function *FROM* which can be used as follows:

```
DISPLAY FROM 'IPS'
```

FROM is now another primitive function of the system. The user was able to build this function without knowing the internals of the system. (Indeed, in such a secure system as an electronic mailbox, he wouldn't be allowed to see inside the system anyway.) The system has now been extended in its capabilities.

Applicability to Arrays

Just as APL primitives can handle scalars, vectors or higher rank arrays, so should applications designed in APL. For instance, consider a function, *PV*, which calculates the present value of a **cash flow** stream, based on a series of **parameters**, such as discount rate and timing of payments:

parameters *PV* **cashflow**

PV should be able to handle a variety of arguments as follows:

- When both **cashflow** or **parameters** are vectors, the single set of **parameters** applies to the single **cashflow** stream.
- When **cashflow** and **parameters** are both matrices, with an equal number of rows, the set of **parameters** in each row is applied to the **cashflow** stream in the corresponding row.
- When either **cashflow** or **parameters** is a vector or one-row matrix, it is extended to match the number of rows in the other argument and applied to each in turn.
- When **cashflow** and **parameters** are matrices with an unequal number of rows (not one), an error results.

Sound familiar? Of course—it's just the APL rule of scalar extension applied to a user-defined scalar function (with a rather loose meaning of scalar). We can even make **cashflow** and **parameters** explicitly scalar by making them elements of an enclosed array.

You should ensure that the functions in your system handle all ranks of arguments, wherever meaningful.

Use of State Settings

APL makes effective use of state settings to specify things that change infrequently, such as ⎕*IO* and ⎕*PP*. These state settings are in a sense arguments to certain primitive functions; however, treating them as state settings, since they change so infrequently, is a great convenience. Open systems can take advantage of the same technique. Such things as:

```
      PRINTWIDTH 132
WAS 80
      OUTFILE 19
WAS 19
      TERMINAL 'DM1100'
WAS HDS108
      DECIMALS 3
WAS 2
      NOTIMESTAMP
WAS TIMESTAMP
      EUROPEAN DATES
WAS AMERICAN
```

are all examples of useful state settings. Each state setting stays in effect until explicitly changed by the user so that he doesn't have to repeatedly state them as arguments to relevant functions. For each state setting, a convenient default must be established. For instance, for *PRINTWIDTH*, a conservative default setting might be 80. When a user first learns the system, he does not even have to know that print width is controllable. But the first time he says "But why can't I get my report printed across the whole of my 132 character page?", he is introduced to the new state setting *PRINTWIDTH*. This is a very desirable way to build a system: you can postpone telling the user about many of its features until he has digested the basics. Just like APL!

A state setting function should return any message as an explicit result, rather than simply as output, so that, if desired, a function such as *WITH* can simply "gobble up" the output message.

```
      PRINT 'REPORT1' WITH PRINTWIDTH 132
```

A state setting function can be easily extended so that a null right argument returns what the state is, without changing it.

```
      PRINTWIDTH ''
132
      TERMINAL ''
DM1100
```

A further very useful enhancement, where state settings must belong to a limited set, is to allow a question mark as a right argument to get information about legal responses:

```
      TERMINAL '?'
SPECIFY A TERMINAL CODE TO TAKE ADVANTAGE OF SPECIAL
FEATURES.  CHOOSE FROM:

AJ832         ANDERSON JACOBSON 832
DM1100        DATAMEX 1100
IBM3270       IBM 3270
HP2641        HEWLETT PACKARD 2641
   .                 .
   .                 .
   .                 .
```

A system which makes use of state settings must provide a mechanism for the user to find out what state settings are in effect, perhaps as a simple function, *STATE*:

```
      STATE
PRINTWIDTH 132
PAGEDEPTH 60
   .
   .
   .
```

This output should be formatted in exactly the form the user would use to reset states, so that it serves as a reminder of the correct syntax.

A final gesture is to provide for the user a simple way to return all state settings to the default state, all at one time. A function to do this could be called *DEFAULT*.

Using state settings implies the introduction of global variables into your system, whose names can potentially interfere with names the user wants to use. To minimize this conflict, minimize the number of names you use: you might save all state settings in a single general array. For the globals that you must create, you run into a conflict. For the sake of maintainability, you want to choose names which are meaningful and adhere to a conventional pattern of name construction. However, to do so is to increase the risk of choosing a name that the user (perhaps an APL programmer himself) might also have in his own system. So steer clear of conventions already used by other packages. This is one time when you're living dangerously if you *don't* break with convention.

Consistency

APL is a highly consistent language. When you have learned the rule of scalar extension, you know it will be applied consistently to all scalar functions. You can depend on edge conditions, such as null arguments, behaving in the appropriate fashion for all functions. Where possible, suggestive symbols have been chosen for primitives, such as ⌊ and ⌈. When you have learned parts of APL, you can sometimes guess at other parts, because you know they will follow the same pattern. What you've already

learned is reinforced every time you add something to your store of knowledge. Users will find it easier to learn your system if you strive for such consistency. The essence of consistency is for the user to be able to predict the behaviour of your system by knowing, or inferring, a small number of rules.

Name construction: In building a system, it is usually necessary to introduce a rather large number of new names, for both functions and variables. It is essential that the **construction pattern** of these names be consistent. If you have functions

```
GETDATA
PUTDATA
LISTDATA
```

then the function that allows you to **change** the data should be called

```
CHANGEDATA
```

not

```
CHANGE
```

If you are building a financial model for capital investment analysis, then the variable names should exhibit a pattern easily picked up by the user, such as:

```
FCPROD1   ⍝ fixed costs for product 1
FCPROD2   ⍝ fixed costs for product 2
VCPROD1   ⍝ variable costs for product 1
VCPROD2   ⍝ variable costs for product 2
```

Within this overall goal of consistency, try also to minimize the number of characters the user must type. However, don't sacrifice clarity to save a character or two. Using *GRAFIX* rather than *GRAPHICS* saves two characters of typing *only* if the user types it right the first time. Otherwise, it increases the number of characters typed in by six.

Function arguments: Consider the primitive functions, ⋆ and ⍟, for exponent and log which are both "base dependent". For both functions, the base is the left argument and the data is the right argument. This same consistency should be seen in your functions. Yet how many times have we seen application systems with irregular commands such as these from an imaginary graphics system:

```
'BLUE' DRAW PICTURE1

PICTURE2 SHADE 'RED'
```

The following commands would be much clearer:

```
'BLUE' DRAW PICTURE1

'RED' SHADE PICTURE2
```

Note that there is a very strong reason, too, for deciding which argument goes on which side. Just as in English we would say "draw a picture", with the direct object following the imperative verb, so the object, *PICTURE*, should naturally follow the function name. Modifiers (or adverbs), such as the colours, should go on the left, because you can then establish a useful default, and allow the user the option of eliding the left argument [Note 2]. In this particular case, *BLACK* can be chosen as the default, and so the user can type:

```
DRAW POLYGON
```

to have the shape drawn in black. Naturally, all functions in the system should conform in using *BLACK* as the default when the argument is elided.

Consistency with APL: Not only should your application system be internally consistent, but, wherever possible, it should be consistent with APL. When the system has features parallel to APL, copy APL! Consider a modelling system which allows APL-like statements to be entered under prompted control to define a "case" of input to the model:

```
PROJECTLIFE←10
COST←10000 20000, (5 YEARS AT 15000), THEN 8000
COST←.10 INFLATE COST
INVESTMENT←40000 50000 10000 THEN 0
   .
   .
   .
∇
```

The "natural" character for indicating completion is ∇.

If you provide a sample line editor, consider / (slash) and ., (dot comma) editing as allowed in immediate execution and function editing in APL. If you provide a function to *SAVE* a case, use the same convention as)*SAVE*. Warn the user if he is attempting to overwrite an existing object while naming as he goes.

As a by-product, a system such as this teaches its users some APL and instills all the right instincts that will be relevant in using APL.

Being "almost" like APL can be worse than being decidedly different. A user could infer that a *SAVE* function works just like)*SAVE*, based on observation. If *SAVE* failed to give the warning before an imminent overwrite, a user who depended on that would be betrayed.

Other Places for Consistency: There are countless other aspects of a system where you should strive for consistency:

- If some functions in your system accept arrays, make sure they all do. If certain functions require matrix arguments, but gracefully accept a vector and interpret it as a one-row matrix, make sure they all do.

- If your system uses name lists (as for example our sample project planning system), make sure you use the same delimiter in all cases.

- There are two common ways of treating state setting 'flags':

HSPRINT ON and *HSPRINT OFF*	or	*HSPRINT* and *NOHSPRINT*

  Choose one style and stick to it for all such state settings.

- If some state settings return a result of the existing setting when fed '' as an argument, they all should.

- If some report functions require RETURN to be typed before a report starts printing, make sure all reporting functions require this.

LANGUAGE DESIGN IN CLOSED SYSTEMS

The design goal in a closed system is to provide help and guidance as needed by the user, and to protect him from errors, while minimizing the restrictions imposed on the user by the system.

We, as APL programmers, find the APL environment extremely hospitable. However, a naive user can be mystified by it. We've probably all seen users who have let their *)SI* stacks grow to dizzying proportions, through lack of understanding of function suspensions. We've seen users dismayed by such "friendly" messages as *INTERFACE CAPACITY EXCEEDED*, *ERR010S ABEND - SYSTEM CODE* - 064, *USER CODE* 0000, or even a simple *RANK ERROR*. A closed system sits between the user and APL, and interprets such messages and restates them in language more easily understood by the user. The closed system serves to insulate the user, for better or worse, from APL.

Closed systems should exhibit the same characteristics as open systems, namely, flexibility and generality, use of state settings, and consistency. In addition, since the APL system itself is no longer the user interface, one must be built. Lastly, an effort should be made to let APL programmers gain access to the system.

Flexibility and Generality

Instead of building functions, now we are building responses to prompts. We want the conversation to be general rather than specific. Take for example the following conversation:

```
COMMAND: REPORT
PRINT FIELDS: SUPPLIER,INV NO,ORDER,AMOUNT
SORT FIELDS: SUPPLIER,ORDER/INV NO
TOTAL FIELDS: AMOUNT

SUPPLIER       INV NO   ORDERED BY  AMOUNT

FLY BY NIGHT   2001     MKTG          1046
FLY BY NIGHT   2012     MKTG           236
                                      ----
FLY BY NIGHT            MKTG          1282

FLY BY NIGHT   1022     PROD           135
FLY BY NIGHT   1045     PROD            37
                                      ----
FLY BY NIGHT            PROD           172
                                      ====
FLY BY NIGHT                          1454
```

```
ON THE SPOT    1882      ACCTG          106
ON THE SPOT    2002      ADVER           37
                                       ====
ON THE SPOT                            143
```

In this case, / (slash) in the response to *SORT FIELDS* indicates that subtotals are to be created for each break in the fields to the left of the slash, but not those to the right. Thus the report contains subtotals of amount for *ORDERED BY*, and for *SUPPLIER*. This is parallel to the generalized *DISPLAY* function of the open project management system shown earlier. Rather than supplying a specialized function, *LISTSUPPLIERSTATUS*, the process is broken down into steps. The user is prompted for the specification for each step. This way, he can specify a wide variety of reports following the same general format.

A simple extension can be provided to vary this general format of the report. By using * instead of /, he can request that certain fields be brought up to the heading.

```
COMMAND: REPORT
PRINT FIELDS: SUPPLIER,ORDER,AMOUNT
SORT FIELDS: SUPPLIER*ORDER/
TOTAL FIELDS: AMOUNT

SUPPLIER=FLY BY NIGHT
ORDERED BY              AMOUNT

MKTG                      1282
PROD                       172
                          ----
                          1454

SUPPLIER=ON THE SPOT
ORDERED BY              AMOUNT

ACCTG                      106
ADVER                       37
                          ----
                           143
```

Note how much these responses resemble a little reporting language with the symbols chosen from the set available to the user with a non-APL keyboard.

By providing a generalized system such as this, you give the user freedom to create a report of his choice with very little effort. It means more work for the implementor at the beginning of the system, but allows the user to be more self-sufficient, thus relieving the programmer of responding to requests to produce yet another canned report. We'll see later how a closed system can provide the ability to write a "program" in this reporting language.

The Conversational Interface

Whereas in an open system the environment and syntax of APL form the framework of the user's interaction with the system, in the closed, this "interface" must be created by the application programmer.

The conversational interface is thus the cornerstone of a closed system. The first requirement for all conversation is clarity. All communication to the user must be phrased unambiguously. Since so many APL timesharing users gain access with asynchronous terminals, often remotely connected from places where bandwidth is expensive, the dialogue should also be terse (both input and output). It is no mean feat to achieve clarity and terseness both at the same time.

For clarity, you must choose your basic terms carefully and use them consistently. For terseness, you should allow the user to type *ALL* as a special case response where he is being asked to enter one or more responses from a list.

You must maximize the system's communication with the user. Special messages are appropriate in several situations in a conversational dialogue:

- An acknowledgement when an update to a file is complete, or an update request has been filed, or a calculation performed. *DONE* is a succinct way to tell the user this. Conversely, if some expected action does not complete, let the user know that it was *NOT DONE* [Note 3].

- A warning and request for confirmation before taking any destructive action, such as erasing or overwriting data, or deleting a file.

- A warning when the requested action would be very expensive, cause lengthy output, or entail a lengthy delay.

- An indication that certain offline tasks have been requested, have begun, or have terminated abnormally (these might include batch tasks, offline printing, remote graphics, etc.)

- An indication of side effects, such as the automatic creation of a file [Note 4].

The most important thing you must do for an effective conversational interface is to formulate a small set of rules, from which the user can predict the behaviour of the whole system, and to embody those rules in APL code.

To do this, you must develop a single mechanism for handling all dialogue. This means the user has a completely consistent mechanism to learn, and can infer much information throughout the exchange. Let us call this generalized interface the *ASK* subsystem and examine its characteristics and behaviour.

ASK **Subsystem** The first duty of the *ASK* function is to keep interaction with the user under program control at all times. This implies that all interaction should be via ⍞, not □. So our first shot at *ASK* could be as follows:

```
    ∇ ENTRY←ASK PROMPT;RHO
[1]   RHO←⍴⍞←PROMPT,': '
[2]   ENTRY←RHO↓⍞
    ∇
```

This is one of the first utility functions a budding APL programmer writes, if only for his own convenience. *ASK* places a colon after the prompt to make it easy to decipher a terminal session, discriminating between output and input. Avoid using colons for other sorts of output to retain the value of this convention in the system.

The user should encounter the same behaviour every time he is asked to choose an option. An *ASKOPTION* function, which calls *ASK*, can handle requests for the entry of a menu option:

```
    ∇ CHOICE←OPTIONS ASKOPTION PROMPT;LENGTH;MATCH;COUNT
[1]  ⍝ OPTIONS ARE IN MATRIX FORM, ONE OPTION PER ROW
[2]  INPUT:CHOICE←ASK PROMPT
[3]   CHOICE←(' '≠CHOICE)/CHOICE
[4]   LENGTH←(¯1↑⍴OPTIONS)⌊⍴CHOICE
[5]   MATCH←OPTIONS[;⍳LENGTH]∧.=LENGTH↑CHOICE
[6]   COUNT←+/MATCH
[7]   →(COUNT≠1)/ERRORS
[8]   CHOICE←MATCH⍳1
[9]   →0
[10] ERRORS:→(COUNT>1)⍴AMBIG
[11]  □←ERROR 'INVALID OPTION ',CHOICE
[12]  →INPUT
[13] AMBIG:□←ERROR CHOICE,' AMBIGUOUS WITH:'
[14]  □←' ',MATCH⌿OPTIONS,[1] ' '
[15]  →INPUT
    ∇

   ∇ Y←ERROR X
[1]  Y←'**ERROR- ',X
```

The *ERROR* function does little at this stage, but is included because we have a hunch that ensuring that all error messages pass through a common mechanism will come in handy later on.

Note that the *ASKOPTION* function only requires the user to type as many characters as are needed to uniquely identify one option; the user has a convenient way of minimizing typed input. You must choose option names with great care for this feature to be of much use to the user. Suppose you are developing a merger analysis package and you choose your option names:

```
COMPARE   A compare before and after restructuring
COMPUTE   A compute profitability indices
COMBINE   A merge an acquisition
COMPRESS  A delete unprofitable divisions
```

The user would have to type at least 4 or 5 characters to distinguish one option from another. On the other hand, by choosing the names:

```
COMPARE
INDICES
MERGE
DELETE
```

the user never has to type more than one character to make this distinction.

We must provide a simple way for a user to determine which options are actually open to him—in other words, a *HELP* facility. At least two levels of help are required in most systems:

- a concise list of possible entries for the experienced user who needs a quick reminder

- a more detailed message, providing some details about each possible choice for the user going through the system the first time

The second-level help messages should be designed so that a user familiar with the application area, but not the system, can find his way through the system alone (well, almost). You should be able to say, "*HELP* yourself".

HELP messages should be dynamic. If a user is asked to enter a field name, and types *HELP*, the message should include a list of all legal field names. This implies that, within the system, *HELP* must be provided through a function, not a static message.

HELP is not the only conventional keyword that must be provided. A consistent way of stating "I'm finished entering data for this prompt" must be provided, whether it be *STOP* or *END*, or a conventional interpretation of SPACE RETURN. Similarly, a keyword such as *EXIT* or *OFF* must be provided to allow the user to get completely out of the system. (In a highly secure system, *EXIT* may not leave the user in a workspace, but rather sign him right off the system.)

Let us for the sake of argument choose the keywords *HELP*, *STOP*, and *EXIT*. We can embody these in our *ASK* subsystem by rewriting *ASK* to look for these special keywords and to return an appropriate signal when found:

```
   ∇ ENTRY←HELP ASK PROMPT;MATCH
[1] START:⍞←PROMPT,': '
[2]  ENTRY←(2+⍴PROMPT)↓⍞
[3]  MATCH←(3 4 ⍴'HELPSTOPEXIT')∧.=4↑ENTRY
[4]  ⎕SIGNAL(1↓MATCH)/ 901 902
[5]  →(~MATCH[1])/0
[6]  ASKHELP
[7]  →START ∇
```

Note that this method means that *ASKOPTION* and the main calling function must be modified to take appropriate action when they detect the ⎕*SIGNAL*s. This method gives complete freedom as to how the *HELP* function is written, so that dynamic help can be provided, and as many levels of help as required can be provided. You will also notice that there is already far more code for handling errors, help, and special cases than for handling the original entry. Don't be dismayed! This is typical of friendly systems: the user interface consumes a nontrivial part of the effort, and the code.

The use of APL systems can sometimes have a wide geographic spread, thus users may have different native tongues. *ASK* can be modified to handle this possibility gracefully. We need only change the basic *ASK* function to expect an argument which is a general array, with each element being the prompt in a different language. A global variable, *LANGUAGE*, contains the pointer to which element of the array holds the prompt in the desired language. So the new first line of *ASK* becomes:

```
PROMPT←⊃PROMPT[LANGUAGE]
```

Since errors and help messages are always treated by the functions *ERROR* and *HELP*, they too can be modified to take into account an enclosed array of messages and the *LANGUAGE* variable.

Even without local language differences, we may want to have two styles for the dialogue: *VERBOSE* and *TERSE*. In the *TERSE* setting, prompts are kept to a very minimum, perhaps in the extreme case to just a colon; you depend on the *HELP* messages to give enough information if the user needs it. The *VERBOSE* setting provides much longer prompts.

It is even better to let the user avoid prompts altogether by letting him enter a list of answers in anticipation of the prompts he knows are coming. These answer lists must be separated by a delimiter such as semi-colon, so that a sequence like the following:

```
OPTION: REPORT
CATEGORY: BUDGET
ITEMS: EXPENSES,REVENUE
DATE: SEPT
```

can be replaced by:

```
OPTION:REPORT;BUDGET;EXPENSES,REVENUE;SEPT
```

The provision of this facility is simple, given the *ASK* subsystem we have already built. We need one global variable for an input buffer, *BUF*, and two new lines in *ASK*:

```
    ∇ ENTRY←ASK PROMPT;LENGTH
[1]   PROMPT←⊃PROMPT[LANGUAGE]
[2]   →(2=⎕NC 'BUF')/L1
[3]   BUF←''
[4]  L1:→(0≠⍴BUF)/L2
[5]   ⍞←PROMPT,': '
[6]   BUF←(2+⍴PROMPT)↓⍞
[7]  L2:LENGTH←BUF⍳DELIMITER
[8]   ENTRY←(LENGTH-1)↑BUF
[9]   BUF←LENGTH↓BUF
    ∇
```

The *ERROR* function requires one final line, so that the *BUF* variable is emptied when an error is encountered.

There are cases where a user will want to make the same choices many times. The *ASK* subsystem can cater to this behaviour by allowing the user to define an answer list (by enclosing it in brackets and to give it a name by setting it off with an assignment arrow. The user can enter the definition of such an answer list as shown below:

```
OPTION: [BUDREP←REPORT;BUDGET;EXPENSES,REVENUE;SEPT]
```

Later, the user can refer to this answer list by name:

```
OPTION: [BUDREP]
```

and it would be equivalent to entering the answer list directly. These **P**redefined **A**nswer **L**ists are so friendly, we call them PALs!

Note the choice of assignment so that a PAL looks like an APL variable. This is another instance of compatibility with APL. It will certainly appeal to the APLers, and will provide a small bit of familiarity with APL for the user we hope to tempt into APL later.

The net result of all this is the *ASK* function shown on the following page.

```
    ∇ ENTRY←ASK PROMPT;LENGTH;PALNAME
[1]   →(0=⎕NC 'BUF')/L1
[2]   →(0≠⍴BUF)/SPLIT
[3]  L1:PROMPT←(>PROMPT[LANGUAGE]),': '
[4]  L2:⍞←PROMPT
[5]   BUF←(⍴PROMPT)↓⍞
[6]   →('[]'≠(1↑BUF),¯1↑BUF)/SPLIT
[7]   BUF←1↓¯1↓BUF
[8]   →('←'∈BUF)/DEFINE
[9]   →(2≠⎕NC '∆',BUF)/ERR1
[10]  BUF←⍎'∆',BUF
[11]  →SPLIT
[12] DEFINE:LENGTH←BUF⍳'←'
[13]  PALNAME←(LENGTH-1)↑BUF
[14]  →(~(⎕NC '∆',PALNAME)∈ 0 2)/ERR2
[15]  BUF←LENGTH↓BUF
[16]  ⍎'∆',PALNAME,'←BUF'
[17] SPLIT:LENGTH←BUF⍳DELIMITER
[18]  ENTRY←(LENGTH-1)↑BUF
[19]  BUF←LENGTH↓BUF
[20]  →0
[21] ERR1:ERROR 'PAL ',PALNAME,' DOES NOT EXIST'
[22]  BUF←''
[23]  →L2
[24] ERR2:ERROR 'PAL ',PALNAME,' CANNOT BE DEFINED'
[25]  BUF←''
[26]  →L2
    ∇
```

Use of State Settings

Just as in open systems, state settings can be very valuable in closed systems. The user might want to "set" the language, the terminal and whether it has an APL character set or not, the page depth, and so on. These are normally set outside the control of the main program, and apply to all interaction within the program, so that they don't have to be reset every time the program is invoked.

Closed Systems in the Hands of an APL Programmer

Using a closed system, insulated from APL, can be one of life's most frustrating experiences for an APL programmer. In fact, many APL programmers would look at the heading for this section and say, "If I had a closed system in my hands, I'd strangle it." But it doesn't have to be like that. The trick is to make sure that the knowledgeable APL user, even though insulated from APL, is not locked out.

One easy method is to expand the *ASK* function to recognize ⍎ and to treat what follows as an APL expression. So a user could enter:

```
DATA: ⍎¯10+.2×0,⍳100
```

in a data entry session, or:

```
COLUMN HEADINGS: ⍎¯1↓REMDUPBLANKS,HEADINGS,'/'
```

in a reporting sequence.

These are features which cater to the APL programmer as a user of a closed system.

But how do you make your system friendly for the programmer using a general closed system as a basis for a customized application? He wants to retain the existing interface and capabilities of the closed system but to add certain new capabilities to tailor it to an individual user's needs. The trick is to make it as easy as possible to embed customized code in the system.

A useful technique is to allow the programmer to build **auxiliary commands**. The system is built so as to look for an auxiliary command file of a certain name; if it exists, the system ties it. At appropriate points in the system, a designated component is consulted for a list of auxiliary commands. These commands are logically merged with the current command list in the system, so that it is transparent to the user that they are different from the base system.

Suppose you have a relational data base system which provides commands for adding, deleting and listing records. A user wants to use it as a basis for a pension administration system which has relations for employees and retirees. When an employee retires, one record must be added to the retiree relation and one deleted from the employee relation, with certain well-defined rules governing the information to be transferred. While the user could use several separate commands to effect the transfer, the system can be made more convenient by creating a customized command called *RETIRE*. The system would now behave as follows:

```
COMMAND: HELP
VALID COMMANDS ARE ADD, DELETE, LIST, RETIRE
COMMAND: HELP
ENTER ONE OF
      ADD         ADD A RECORD
      DELETE      DELETE A RECORD
      LIST        LIST RECORDS
      RETIRE      RETIRE AN EMPLOYEE
```

To the user, the command *RETIRE* appears no different from the other three. If the user choses *RETIRE*, the software to be executed is brought in from the auxiliary command file, including all associated communication in the form of prompts, helps, and error messages.

For a programmer to make effective user of the auxiliary command file concept, you must provide him with sufficient system documentation. For instance, you must indicate how to integrate your *ASK* subsystem and document such global variables as *LANGUAGE* used above. You must also document the mechanism for dealing with the data base. If you've built the open system properly, it should suffice as a useful set of tools.

There is one thing to watch out for: the base system may someday be extended to include an option named *RETIRE*. To protect against this, an auxiliary command always takes precedence over a base command of the same name. This avoids trauma to the users. (It's also a useful testing aid as discussed in the next section.)

ROBUSTNESS IN APPLICATION SYSTEMS

Most systems tend to work under favourable conditions and expected events. For a truly friendly system, it is the programmer's duty to ensure that it is robust under unfavourable conditions. There are several classes of unexpected events: silly user input, line drops and system crashes, and (heaven forbid!) program bugs. The techniques for handling these are, respectively, validation, restartability, and bug files. The last step is to test that the software works, and provide a path for modification to ensure that it continues to work.

Validation

Implicit in all our discussions has been the requirement that the user be protected or cushioned when he makes an error. We'll now examine how to deal with that assumption.

We have already shown how to handle errors which can arise when a user is selecting an option from a menu. In a data entry module, errors can arise in several ways, and you should provide methods to validate the data against each possible type of error. There are six main types of validation. Two of these check for the *form* of the data entered:

- Type validation: is this entry the right type? e.g., all character, or all integer?
- Format validation: is this entry in the proper format? e.g., *X*9*X* 9*X*9 for a Canadian postal code? For purely numeric data, format validation takes the form of checking how many valid numbers have been entered.

Four types of validation check for the *content* of the data entered:

- Range validation: is this numeric value in a specified range? e.g., positive?
- List validation: does this entry appear in the list of legal entries? e.g., is it an existing budget category?
- Comparison validation: is this value realistically close to a previous value? e.g., is the day's change in currency rate less than 10%?
- Expression validation: do the numbers entered satisfy a given expression? e.g., do the first ten numbers sum to the eleventh number?

You should apply as many checks as possible to any data entered into your system, for your system will only be as good as the data in it. You can write the most elegant analysis functions in the world, but if the data is invalid, so is the analysis.

You can assist your users by providing tools to help with validation. You could write a very nice collection of error checking functions for each of the six classes of validation:

R← **type** *TVAL* **data** ⍝ where **type** can be *'CHAR'*, *'NUM'*, *'INT'*, *'BOOL'*, *'FP'*, or *'COMP'*

R← **limit** *RVAL* **data** ⍝ where **limit** is **lower limit**, **upper limit**

R← **list** *LVAL* **data** ⍝ where **list** is a table or a delimited vector

R← **format** *FVAL* **data** ⍝ where **format** describes the legal format, using *X* for characters, and 9 for numbers

R← **change** *CVAL* **data** ⍝ where **change** is the maximum reasonable change, in %

R← **expression** *EVAL* **data** ⍝ where **expression** is a valid APL expression which must be true for the *DATA* like *DATA*[11]=+/*DATA*[⍳10]

These functions should generate appropriate error messages as an explicit result, or an empty vector if the data is valid.

These functions can be easily offered to the user of an open system. In a closed system, you should provide a way for users to insert their own validation routines (in large multi-user data base systems, standard validation functions might be inserted by a steward, or data base administrator). Without having to know APL, a user could choose his own validation routines. The validation definitions for a personnel system might look like this:

```
ENTER VALIDATIONS TO BE DONE:
'(999) 999-9999' FVAL PHONE
JOBLIST LVAL JCODE
16 65 RVAL AGE
5 CVAL SALARY
STOP
```

For efficiency reasons, it may be desirable to "turn off" or at least postpone validation. You could let your users specify whether validation is to be done entry by entry (most expensive, and most friendly), once per session, or postponed to be done as a batch task offline (least expensive, but least friendly).

An open system has to validate more than just data entry. Functions in an open system should check that the arguments of the function are legal arguments, and should return an error if they aren't. Care should be taken to have errors occur at the correct level of the stack. For instance, suppose you write a permissive *PLUS* function, which "extends" a vector (⍵) along the appropriate dimension, and adds it to a matrix (⍺).

When there is a length error due to the arguments not matching, we exit *PLUS*, without having set the result *R*. The error interrupts the calling function with a

RESULT ERROR (or *VALUE ERROR* in some systems). This message does not convey the right sort of information back to the user. Causing the function *PLUS* to suspend is no improvement. That simply means the user sees the error in *PLUS*, when really the error was in the calling function of *PLUS* which provided incorrect arguments. This can be handled by writing an error line:

```
ERROR: 'UNMATCHED ARGUMENTS' ⎕SIGNAL 999
```

This results in the error appearing as follows in a user's function:

```
UNMATCHED ARGUMENTS
DO[23] CALC A PLUS B
           ^
```

Now *PLUS* appears to the user to behave just like an APL primitive function, and delivers a meaningful error message.

If such an error may have to find its way through several levels of your functions, you can prevent *UNMATCHED ARGUMENTS* from "popping up" in any level but the user's function by setting a trap:

```
⎕TRAP←'∇ 999 C (5↓⎕ER[⎕IO;])⎕SIGNAL 999'
```

in each top-level function which could receive such a signal. This trap definition cuts back the execution stack and reissues the error to the open workspace.

A further elaboration is to provide a set of signal numbers that tell the user of your system both that the error came from your system and its specific interpretation. The user's package can then exploit this information. You must document for the user the meaning of these error signal numbers.

There is one notable situation in which it is desirable to exclude error checking from the system, in order to keep costs down. This approach is appropriate when you anticipate that the system's functions will only be invoked by the functions of a higher level system. In such a case, error checking is done by the higher level system anyway, and the user should not be penalized by having it done in the kernel system as well. It is better to assume that the higher level system will always be passing legal arguments to the kernel system. If the system is to be used in two different ways, as a user system, and as a kernel system, the checking could be switched on and off according to the type of use.

Restartability

Your program has to withstand the vagaries of computer system crashes, line drops, or the mad attention-key fanatic. The system should be able to gracefully restart after such abrupt and untimely interruptions.

The answer is a *RESTART* function which allows the user (or the local ⎕*LX*) to execute

```
→RESTART
```

to resume execution. *RESTART* should

- warn about multiple suspensions
- retie previously tied files
- execute user profile
- re-establish shared variables
- restore session parameters
- re-initiate any "slave" background tasks
- calculate the line number at which to restart

In many systems, the restart line number is simply ⎕*LC*. In systems with sensitive code, the calculation of the restart line may have to be more complicated to ensure restart from a safe place.

In order to be able to tie exactly those files needed, you will need to keep track of what files are tied. You will have to write cover functions to tie, rename, erase, create and untie files, which take a copy of ⎕*NUMS* and ⎕*NAMES* and store them in global variables, as in:

```
      ∇FNO←STIE FNAME
[N]   ...⎕STIE...
[N+1] FNUMS←⎕NUMS ◇ FNAMES←⎕NAMES
```

Similarly, you'll have to keep track of shared variables, and session parameters so that they can be re-established. You have to write code that is in fact restartable.

In a system where you have to provide a restart capability for the user who doesn't have an APL keyboard, and so doesn't have the character →, you can contrive this by writing a function *RESUME*:

```
    ∇ RESUME
[1]  ⍝ DEFAULT GLOBAL  ⎕TRAP←'∇ 888 E →RESTART'
[2]   ⎕SIGNAL 888
    ∇
```

Bug Files

Of course, you hope that your application system is completely bug free. However, this is never the case. So you must prepare for the worst and provide a mechanism which protects the user from being suspended, makes you aware of the error, and provides information to help you fix it. A bug file generated by a trap mechanism answers all these needs.

Event trapping makes it possible to prevent *all* unanticipated program errors from reaching users. A trap can be set at the topmost level of the system (remember an open system has many topmost levels) that catches any unanticipated error, files a report, and issues the user a comforting message:

```
PROGRAM ERROR HAS OCCURRED AND HAS BEEN REPORTED
```

Your users never experience a suspended function and can return to the beginning of the system to restart (or the trap itself can get them restarted).

To file a bug report is to store on file useful information on the state of the workspace at the time of the anomaly. Such information can be collected in a package, which could contain:

`⎕ER`	the `⎕ER` of the event that caused the interruption
`WSID`	`2 ⎕WS 1` at the time of the event
`SI`	`2 ⎕WS 2` at the time of the event
`TASK`	`⎕RUNS [⎕IO;]` at the time of the event
`NAMES`	`⎕NAMES` at the time of the event
`NUMS`	`⎕NUMS` at the time of the event
`VARS`	a package of the variables which are referenced in the suspended line

The user's account number is automatically stored as part of the component information in the file.

Check the bug file often. You should write a bug file management system, which makes it easy for you to view the contents of the file, and to delete bug reports from the file as you handle them.

Testing

The best friend is a faithful friend. The best system is a dependable system. The only way to be sure your system is dependable is to do extensive testing. This testing must be done rigorously and completely—preferably by someone other than the original author of the software. This is a very good task for users as it further increases their involvement with the system. If such a tester is not available, try if possible to leave your system for a couple of weeks, and come back to it fresh to do your testing, for it is very difficult to do a reasonable test when you are thoroughly entrenched in those habits which the system is designed to support. It is the unusual, imaginative, and unforeseen behaviour that causes problems.

It is useful to create a bank of test programs, or a "script", which exercises the system fully. This can then be run against new versions of the system to confirm that they continue to work unchanged, except of course where those differences are deliberate. For a file-based system, this entails the creation of a set of sample file data, which serves another useful purpose for documentation, as described later.

Auxiliary command files provide a very convenient way for new commands to be tested. The implementor can install new commands as auxiliary commands and then ask certain people to test the new code. These people simply tie the auxiliary command file whenever they use the system; the new command is transparent to everyone else. In this way the command can be tested as it fits with the rest of the system; it is usually the subtle interactions between separate parts of the system which cause the most trouble.

Releasing New Versions of Software

Now you have built this marvellously successful system; it grows and changes and adapts dynamically to your users' needs almost before they can articulate them. How can you manage this wriggling beast? Your task will be easier if you can avoid "creeping" changes by collecting bundles of enhancements and releasing them all at one time. Fix a solid reference point in your shifting mass of software, "freeze" it, and give it a version number.

You must also ensure that the new version is upwardly compatible with the previous version. There are three requirements for compatibility:

- The new system continues to work on previously stored data.
- Existing programs continue to work—either the ones using your open-system modules, or the ones embedded within your closed system. Predefined answer lists are types of programs: remember that revising the prompt sequence invalidates them. This is very hard indeed to test, because you aren't necessarily even aware of all the programs written using your system. Most dangerous are those users who have made use of quirks that you consider bugs and want to eliminate.

- User behavioural patterns can remain unchanged. This is a relatively weak requirement from a technical viewpoint but is very critical politically. The user should be flexible enough to learn new habits, but that is not always the case.

The script that was developed originally for testing, and the sample data or data base turn out to be of great value here. If you have achieved upward compatibility, the results of running the original script should be identical with the original release.

Next, you must devise a scheme for formalizing and distributing versions of software. When you release a new version of software, you want to ensure that all users receive the new software. Your system should notify the user which version of software is active. In a system where the functions are all on file and are paged in as needed, this is relatively easy to achieve—every time a user reads the function to be executed from file, he reads the most recent version. In a data base system, you should always go further and identify each data file with a version number, or date, so that a simple comparison can verify whether the software and the data files are compatible in the event of a redesign of the file structure. Then the user can be advised that a conversion program should be run on his data file, or that one is being run automatically on his behalf.

You should always keep a back-up copy of the previous version of your software, in the unlikely event that you need to "back off" the new release. This may soothe a lot of ruffled feathers should any significant errors rear their ugly heads! You can even go so far as to provide an option in the system to let the user choose the version of his choice.

It is more difficult to effect this smooth distribution of new software in a system where the functions can all reside in the workspace, and are not read in from file as needed. The user can simply save your system in a workspace of his own and be stuck with that version forever. One solution is to build an empty workspace in which the $\square LX$ executes *GETSYSTEM* to bring in all your software. This makes the user more conscious of the problem so that he will include *GETSYSTEM* in his $\square LX$ if he embeds the system in a new workspace.

TERMINAL CONSIDERATIONS

The behaviour of terminals can vary a great deal, both in the variety of features offered, and in the commands required to "drive" these features. You should write your systems so that your users can take advantage of as many features as possible without being required to learn the mechanics of their terminals; the system should still run, though undoubtedly less elegantly, on a terminal devoid of features.

- APL or no APL keyboard.

  - A function can be used to translate all dialogue. It is necessary to convert all input and output to and from the closest equivalent APL characters. For instance, non-APLers might be instructed to type = rather than ← for defining PALs (Predefined Answer Lists). A function would translate = to ←. Similarly, if a report used | (stile) as a separator on reports, the function would replace this with !, the closest alternative visually. Thus, a function must be inserted into the *ASK* function to treat all input, and a function must preface all output produced.

  - Where you want users to be able to use APL primitives, such as in ⍝ mode in closed systems, or just in the workspace in open systems, you have to provide a set of cover functions for each of the APL primitives. (There are several sets of such functions available to choose from.)

    These concerns are valid also for a full screen device.

- Print speed increases:

  - Where possible, set the terminal to run with *NOIDLES*.
  - Set appropriate tabs on the machine.
  - Set form feed for terminals, which have such a feature.

- Size of output

  - All report output should be sensitive to the maximum width and depth of the paper.

- If the printhead obscures the printing on a hard-copy terminal, make the *ASK* function sensitive to this by appending a linefeed to all prompts.

These features should all be transparent to the user, who merely sets a state such as *TERMINAL*:

```
TERMINAL 'DECWRITER'
```

From that point on, output and input should be sensitive to the type of terminal.

Full Screen Devices

Designing systems for a full screen device merits some special consideration. When a block-mode full screen device is connected locally to a computer, you can disregard the minimization of characters. Instead, the fixation shifts to the frugal use of the physical space on a screen.

Screen design for data entry requires the same degree of attention as the composition of prompt dialogue. The physical arrangement of the screen can be exploited to draw attention to certain areas, and to de-emphasize fields of lesser importance. Related data should always be displayed or requested on a single screen. For instance, you should use a single input form for a related set of data to fit on one screen and where possible, make the screen like the data input form from which the data derives. All related concepts in *HELP* messages should fit on a single screen. You can use the physical arrangement of a "form" to help in communication with the user. There is a limit, of course, to how much data can be visually distinguished on a form; make sure that you're not trying to squeeze too much onto one screen. Assign a function key which lets the user move from screen to screen. In a many-screen system, you can allocate a field where the user can type in the number of the screen form he wants to enter next.

Just as with line-by-line terminals, a consistent approach benefits the user. A full screen ASK subsystem can consistently take advantage of certain features of the terminal. Error messages should always be treated consistently: reserve a particular part of the screen for displaying error messages. Use colour, reverse video, or flashing to highlight and bring attention to the error messages. This is much more effective than the simple ** used on the line-by-line terminals.

Data input can be reduced by taking advantage of programmable function keys on the keyboard. Here again, conform to established conventions. For instance, use function key 1 to request *HELP*, key 3 for *EXIT*. Some terminals automatically show a user-friendly display of the meaning of each function key in a reserved display area of the screen. Such a feature can make it reasonable to change the interpretation of the program function keys (except the standards) in different parts of the system.

You can also exploit special visual features of screens, such as blinking, reverse video, and colours to make data entry easier and more accurate. You can imply a great deal of information by such techniques as:

- marking fields, say by underlining, to show maximum field width
- using reverse video, higher intensity, or different colours to distinguish fields that are required from those that are optional
- "flashing" errors with the blinking feature or special colours
- filling in fields to illustrate the format. For instance, one might display *DD/MM/YY* to help the user fill in the date in the correct format.

You can minimize data input by presenting the screen to the user with default data already in place. So if the user's account number belongs to the Production Department, the code for that department can appear in the department field every time data entry is invoked. In unusual cases, the user simply overwrites that field. If a user is entering many "forms" of data for which the values for a number of fields do not change for several forms, you can provide a facility for having the form "primed" with all the data from the previous form entry.

Graphics Devices

One graph is worth a thousand numbers. In almost any application area, the provision of graphics display enhances the value of the output, so make it easy for your users to plot their data: write graphics routines yourself, or, better still, integrate an existing graphics package if you are lucky enough to have one on your system.

The graphics system should be able to produce graphical output which elucidates the interpretation of the data. For a business application, this might mean line graphics, bar charts, histograms, or pie charts. For statistical applications, it might be more important to provide scatter plots and line graphs. Some scientific applications might be best served by a surface plot, and the provision of log plots might be very important.

In all cases, the commands to produce the graphical output should be in the language of the application, not the arcane requirements of the hardware device being used. A generalized graphics package should work on whatever assortment of terminals are used by your users. A typical graphics system offers a range of basic functions such as *DRAW*, *SHADE*, and *TEXT*, which work on a variety of devices. By setting

```
DEVICE 'HP7221A'
```

for instance, the user causes all the appropriate functions to materialize to "drive" an HP7221A. These functions constitute a **device driver**. The different device drivers consist of completely independent sets of functions, but, to the user, they look the same.

Once such a set of basic functions is created, any graphic system can take advantage of them, and so can easily be device "independent".

FURTHER USER CONSIDERATIONS

There are several more ways to make your system friendly to users: use of a profile, provision of documentation, and the question of outward compatibility.

Profiles

Both open and closed systems can take advantage of state settings to make things more convenient for the user. But it is often the case that a user wants to have the same states set for every use of the system—or at least most of the time. For instance, the state:

```
NOAPL
```

which indicates that the user is working on a terminal without an APL character set is unlikely to change (it is unfortunate but true that a terminal seldom sprouts an APL character set overnight). But since the default state setting is *APL*, the user has to remember to type *NOAPL* every time he loads the system. A personal profile feature saves this effort by setting a number of states in the *⎕LX* of the loaded workspace or at the beginning of the main function. The user can override the state on the rare days where he gains access to an APL terminal.

Terminal specification is a very common profile setting. Another is to automatically set states which control the system. For instance, a workspace documentation package, WSDOC, might offer a wide range of output—variable listings, function listings, a tree diagram of calling structure, etc.—from which a user can choose. Each user of such a package probably has a favourite "style" of WSDOC such as:

```
LIST FNS,VARS,WSXREF,WSGRAPH
HIGHSPEED
NOPRINT
```

With automatic profiling a user stores these settings as part of his profile. The *WSDOC* program then refers to these settings from the user's profile. Again, the user can override these settings.

User profiles can be used to establish certain defaults, say, for data entry. In a full screen system, these profile defaults would automatically appear in the appropriate positions on the screen. To override them, the user simply types over them.

User profiles can also be used to designate what auxiliary command files are to be tied on behalf of a user.

Documentation

No discussion of user-friendliness would be complete without a section on that bane of the programmer's life—documentation. For there will be one or two cases in the life of a system when a user really will follow the dictum "When all else fails, read the documentation".

When writing a manual, just as with the system itself, you must be very clear about your audience. Is this a reference manual for experienced users? Or is it a tutorial guide for first-time users? It is important not to try and address both audiences simultaneously. The experienced user needs a Reference Manual, structured to make it easy to look up particular topics. It is relatively terse and quite rigorous. The first-time user needs a Users' Guide, structured to be read from front to back. Its emphasis is on explaining in a tutorial way. Completeness and rigour may be sacrificed to clarity and ease of understanding. Sometimes you can satisfy both audiences in one manual by making the Reference Manual into an Appendix of the Users' Guide, probably in the form of a concise function-by-function reference.

You must be very clear about the concepts and terms involved in the system, and explain them early in the manual. These terms must be suited to the application area, not to the underlying computer implementation. You must be sure to use the same vocabulary in the documentation as in the system. If a prompt in the system asks for:

REPORT SPECS:

there is no point in referring in the manual to forms definitions—the two must always agree.

In a tutorial manual, introduce your system intuitively—show what it can do by example first; explain later. The best way to write a manual is to use examples as the basis of the manual and to weave the text around them [Note 5]. Some users will simply scan the manual until they find examples which resemble what they want to do. Here again your attention to consistency will pay off: by looking at one or more examples, they should be able to guess how to do what they want. Sample system data should be created so that the user can reproduce with real data the examples in the manual. This sample data can be very convenient for those charged with setting up training courses.

You should avoid the tedious explanation of a lot of special cases and aim instead to explain the general rules. Explaining these general rules is a healthy discipline for the programmer. If these general rules can't be explained briefly and clearly, then the programmer should look to the system to try to correct the flaws. This will start another healthy cycle in system development: development, documentation, development, documentation, development, and so on.

In helping to build a user-friendly system, and keeping your users happy, it can be very valuable to have the users participate directly in the writing and editing of the manual.

Besides the manual, it is necessary to have online documentation. In some simple systems, online documentation may be sufficient by itself. The essence of online documentation is brevity. You want the user to be able to print out the information he requires as quickly as possible. This implies writing the documentation tersely, and breaking it up into many modules, so the user doesn't have to print a great mass of documentation before reaching what he wants. The information should be arranged with the most important facts first so that a user can "break" off once he's read what he needs to know. *DESCRIBE* is, by convention, the place users look for online documentation on loading a workspace. *DESCRIBE* should explain what the system is about and point to what other documentation is available.

Online documentation should be arranged so that it can be printed at the terminal or at a highspeed printer; and there should be a way to get *all* the documentation with one command. You should also minimize white space, ensure that the documentation fits within 80 characters and makes sense even on a terminal not equipped with an APL character set, and eschew underlining.

An online reference card is helpful. It is intended to give a quick aide-memoire to those who already know the system and just need a quick refresher on the details in order to be able to use the system. It simply lists the names of all functions, with their syntax and a description of the arguments. The *DESCRIBE* should tell the user about the existence of the online reference card.

Another feature to provide in online documentation is a *NEWS* system. This is a way to disseminate information about new features of the system. Such a news system should keep a complete record of all news items and inform users which items they have not yet read.

There is another type of documentation which has been entirely ignored here, and that is system documentation. This topic is better explored R. Metzger's paper, "Designing Maintainable APL Systems". But it should be pointed out that there is one case where the provision of good system documentation is a requisite of user-friendliness. This is the case where your system provides for auxiliary command files. The user is a programmer and needs adequate documentation to exploit the feature you have provided him. So, besides the Users' Guide, you need to provide a Programmers' Guide to the system.

Outward Compatibility

No system is an island. You must ensure that your system is compatible with other systems in your environment. This compatibility should be at several levels:

- All terms and concepts should be compatible—if you work with timeseries, for example, make sure the term means the same as for other systems.

- Formats should be the same: Date formats are a case in point—make sure they are the same as for other systems. Name lists are another: decide

whether comma, blank or some other character is to be the delimiter in name lists based on what other systems do.

- Avoid name conflicts with other systems—choose a pattern for your own names and make sure it differs from other systems [Note 6].

- Use the same *ASK* function, or at least one that delivers the same behaviour, as other systems. Choose the same delimiter for answer lists as other systems use.

- Use similar data structures where possible. If other systems all put fields in columns and records in rows, follow suit.

This approach provides a couple of advantages. Firstly, it lessens your own work load, because you can deliver features to your users by "piggybacking" on other systems. Secondly, it lowers the learning threshold for your users because they already know a great deal about your system if they have already used some other.

CONCLUSION

Any successful system, that is, one that gets heavy use, will engender many requests for enhancements and extensions. You must listen carefully to your users, and encourage user feedback. In fact, you should create an explicit channel for such feedback. If you are located near your users, drop in on them once in a while. If you're further away, use the computer to collect these comments. An electronic mailbox is a natural way to provide this ability. For each application system, you can establish a query group, whose member(s) are the developers of that application. In the absence of such a mailbox, a "suggestion box" facility could be implemented.

Once you have garnered all this feedback, you must assimilate and analyze it before embarking on enhancements. Use the 90% rule again—don't implement the feature unless 90% of the users will use it. Be on the lookout for specific requests from different users which can be solved by providing a single more generalized feature. It can be very tempting at this point to abandon the search for generality and consistency and to start adding features willy-nilly. You must be responsive, but you must be discriminating too.

With these remarks, we've come full circle to our starting point in this paper—the APL way of designing and building systems. With APL, you can afford the time it takes to make a system friendly.

All in all, the task of designing a user-friendly system is a challenging one, and highly rewarding. I hope this collection of ideas has stimulated a few ideas of your own, and will make your next system a little bit friendlier.

NOTES

1. Because of this dialogue, this sort of system is sometimes called conversational.

2. R. Metzger's paper, "Designing Maintainable APL Systems", refers to these as "control knobs".

3. Such confirming messages may be very important when a user needs to perform a post mortem to determine exactly how he arrived at a given situation. In extremely sensitive cases, such information, along with all the data input and produced, may be logged to a file, to provide an audit trail for later examination.

4. When such files are created automatically, choose a file name which identifies the system it is associated with.

5. A manual requires "testing" just as a system does. The example output should be inserted under program control. Since a system often evolves during the course of the writing of the documentation, a final test must be done to ensure that the manual examples faithfully mirror the behaviour of the system.

6. You could consider designing your system so that it can run in an isolated workspace and communicate with the user workspace via shared variables. (In SHARP APL, this isolated workspace is an N-task workspace.) This avoids potential name conflicts and provides a sanitary way to interface two systems.

REFERENCES

Berry, P., J. Bartoli, C. Dell'Aquila, and V. Spadavecchia. *APL and Insight.* APL Press, 1978.

Iverson, K. "Notation as a Tool of Thought." *A Source Book in APL.* APL Press, 1981

Martin, J. *Design of Man-Computer Dialogues.* Prentice-Hall, 1973

Smith, A. *APL A Design Handbook for Commercial Systems.* John Wiley & Sons, 1982

APPENDIX

Check list

Evolutionary design process?
Open system implemented first?
Functions have arguments and results?
Few and general functions?
State settings used where appropriate?
- Can you get a list of all state settings?
- Can you find out what a particular state is set to?
- Can you get information about state settings?

Is the name construction pattern consistent?
Are the arguments to functions consistently applied to right or left?
Are errors signalled back to the user level properly?
Closed system implemented using the existing open system?
Are options general rather than specific?
Is all conversation handled consistently by an *ASK* subsystem?
- Can *ASK* handle *HELP*, *STOP*, and *EXIT*?
- Can *ASK* accept option entries which are typed in just for uniqueness?
- Are there adequate levels of *HELP*?
- Is there a capability for foreign languages support if needed?

Are there warning messages for destructive action?
- Can you use answer lists? predefined answer lists?

Are the features of a video screen used to advantage?
- Are state settings used where appropriate?

Can APL programmers conveniently modify the system? Use the system?
Does the system take into account the terminal on which it is running?
Can the user set a profile?
Does the system run? *Always*? You're *sure*?
Can you restart the system after crashes?
Is there clear identification of versions? Have you planned the process for releasing and distributing new versions?
Is there a bug file mechanism in place?
Is the user documentation written? Manual and online?
Is a *NEWS* facility in place?
Are the concepts and terms in the system clear? Consistent between the system and the documentation?
Is the system compatible with other complementary systems?
Is there a channel for user feedback?

DESIGNING MAINTAINABLE APL SYSTEMS

Robert Metzger
Education Manager
I.P. Sharp Associates, Inc.
Rochester, New York

Programmers don't like doing maintenance. They seem to dislike it almost as much as they seem to dislike doing documentation. This is true regardless of the programming language being used, including APL. The purpose of this paper is to explain some of the many techniques that APL programmers can use to design their application systems to be easier to maintain.

To start, we have to think about why we do program maintenance. It is needed for three main reasons. Often, the user of the application is satisfied initially with the way it works. But when he or she sees how the computer can be used to help solve problems, lots of new ideas result. Sometimes he requests additional features. At other times, he requests changes in the way the program works. And, rather infrequently, features may even be deleted. All this activity is called enhancing the application.

The second reason is because the problem that was solved sometimes changes. The economic environment, the nature of the business, etc., can all change. When these sorts of things change, so do the needs of the people who use the system.

The third reason maintenance is needed is as an extension of the development process. That is to say, it was never done right in the first place. Even this situation can have at least three causes. If the behaviour of the application was never formally specified, there is no way of really knowing whether the program was done correctly, until the user uses it. Even if it was specified, the programmer and user may have interpreted the specification differently. Or, most commonly, the program doesn't work the way it should because no one ever thoroughly tested it to see if it did.

So, for whatever the reason, when you maintain software, you change it in order to change its behaviour. This is quite different from maintaining hardware (of any kind). When you do maintenance on your car, you replace or fix or clean parts which have deteriorated through wear. These parts were designed for a specific function by an engineer. If they were manufactured correctly, they perform to a given tolerance level. If people maintained cars the way they maintain software they would be modifying the transmission to also serve as a stereo speaker.

Having decided why we maintain software, and what we want to accomplish, we need to define how we actually do it. When we do maintenance, we:

1) Diagnose
2) Modify
3) Validate
4) Distribute

When you diagnose, you determine what needs changing. When you modify, you change program statements or data. When you validate, you ensure that what you wanted to change has changed, but that nothing else has. When you distribute, you make the new programs or data available to the users.

Different aspects of a system influence the four steps of maintenance in different ways:

1) The locations, quantity, and quality of program statements and data values that the programmer must inspect to diagnose the problem.

2) The quantity and quality of the program statements that the programmer must modify to produce the new behaviour.

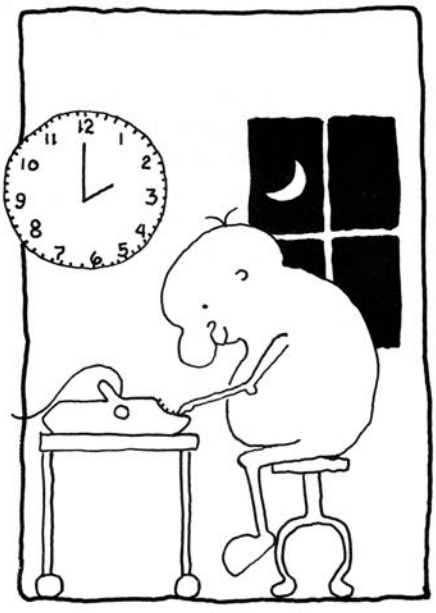
3) The quantity of data that the programmer must look at to validate that the new behaviour is happening, and that nothing else has changed.

4) The number and location of users the new software must be distributed to.

Program maintenance is a hot topic of discussion in the data processing business. Everyone likes to quote horrifying statistics about the percentage of time that professional programmers spend maintaining existing programs. I feel that these statistics are not as relevant in the APL world. This is due both to the type of applications developed in APL, and the type of people who program in APL. APL applications are frequently one-shot analyses, applications prototypes, etc. These require less maintenance because they have a shorter life. Most people who program in APL are *not* professional programmers. Their programming activities support their professional responsibilities.

Nevertheless, the percentage of time APL programmers spend doing maintenance is still significant. If you don't believe it, keep a log of your own APL programming activities for a month. You'll be amazed at how much time you spend changing existing programs. The techniques suggested in this paper will help you to design systems which are easier for you or someone else to maintain, and thus require less maintenance effort.

This paper contains a number of specific recommendations on how to make APL systems more maintainable. Yet, the most important thing you should learn is not a specific technique but rather a general attitude. That attitude can be summed up in the phrase "Premeditated Laziness". What we mean by that is that you should design systems in such a way that when the inevitable maintenance requests come in, the amount of work you must do is minimized.



A different way of saying the same thing is to say that you should follow the Programmer's Golden Rule.

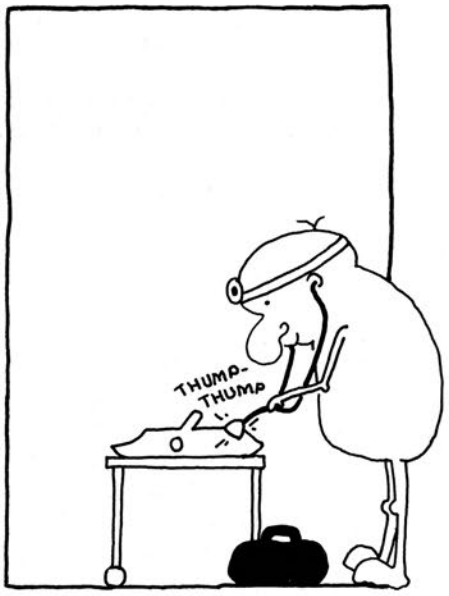
> Write every program as if you were going to be called into the office at 2 A.M. to fix it.

Being considerate to the person who will maintain your program is both altruistic and selfish. It is altruistic because you put in a little extra effort in order to make someone else's life easier. It is selfish because the "someone else" who must maintain your system will often be you!

Start practising a little Premeditated Laziness and follow the Programmer's Golden Rule. The ideas which follow will show you how.

DIAGNOSIS

We want to minimize the effort required to determine what we need to change. In general, the fewer program statements we must inspect, the less effort required. This is subject to one qualification. It is easier to inspect statements if they are all in one program, or in a few programs, than if they are scattered among many programs.



Location and quantity of statements are not the only determiners of diagnosis effort, however. The quality of the statements is just as important. In this context, quality refers to the "meaningfulness" of the statements. There is a spectrum of meaningfulness, from "transparent" to "opaque". A statement is transparent when it is obvious to you *why* a given operation is being done. In contrast, a statement is opaque when you have no idea *why* a given operation is being done.

A seemingly simple statement may be opaque if you can't figure out why it is being done. A very long statement containing many functions may be relatively transparent. The quality of a statement during diagnosis is based on the meaning to the reader. *How* a statement works (as opposed to *why*) can always be determined by executing it at a terminal. We presume that the person doing the maintenance is capable of doing that. If the person doesn't understand how scan works, for example, that is a problem of education, not system design.

The sections which follow will discuss the following topics as they relate to reducing the quantity of statements which must be inspected during diagnosis, as well as to the quality of those statements:

1) Documentation
2) Subfunctions, modularity, and special features
3) Flow of control
4) Naming conventions
5) Statement style
6) Red herrings

As you can see, there are lots of techniques for making the diagnosis process easier.

Documentation

How can you reduce the number of places someone must look in to diagnose a correction or enhancement? The first step the maintainer takes should be the last you took—documentation. The system documentation should be a road map to the application. It should help the programmer to connect behaviour (dialogues, reports, etc.) with programs and data structures.

Your reference documentation should begin with an explanation of the fundamental terms and concepts of the application. The person who must maintain your programs after you may not be nearly as familiar with the application as you have become.

Next, the relationships between data (data structures) also need external descriptions. This information must be written down by the designer since it cannot be generated by a program. Explanations of structures both in the workspace and in files can be stored online for easy reference. A common convention is to store a description of the structure of a file in its first component. The most important part of such documentation tells how to traverse the structure to find a given data item.
Finally, you must document the relationships between functions and variables. If you are using SHARP APL, for example, the WSDOC package can help you produce such documentation automatically. Two features are particularly helpful here. The WSGRAPH report prints a tree structure of function calls. It enables you to see at a glance all the functions which are directly or indirectly subordinate to a given function. The WSXREF report prints a list of every identifier used in the workspace, and every function it is used in. Together, these two reports help you to easily find subfunctions and calling functions which might be affected by a change. Of course, such information can be prepared manually as well.

Internal documentation can help as much as external documentation. You can comment sections of your programs, summarizing what they do. This can help the maintainer find the section that needs changing. This assumes that the comments are kept up to date. It also assumes that the comments are more than just a restatement in natural language of what the APL obviously says. The statement below is an example of useless rehashing.

```
[⍳] Z←X÷Y ⍝ DIVIDE X BY Y
```

Comments about a single line of code sometimes reflect the guilty conscience of a programmer who has used some system-dependent feature or side effect.

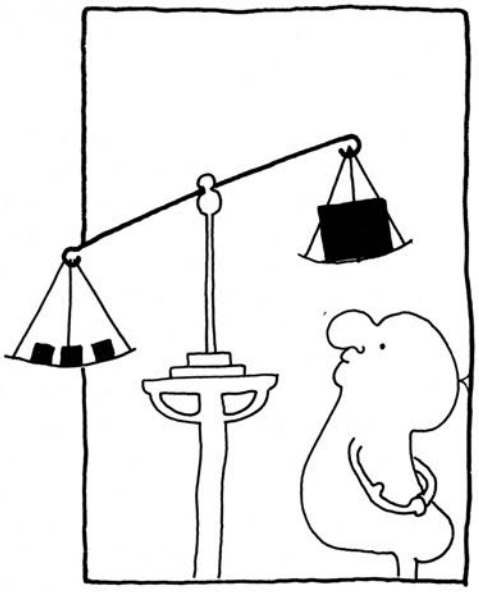
```
[⍳] N←⎕PNAMES ⎕PACK N ⍝  VECTOR NAMELIST TO MATRIX
```

Such comments, by themselves, are of limited value.

The comments which help the most in diagnosis are those which explain the purpose of a section of code. The purpose should be stated in terms of the application. This helps the maintainer to relate the program to the problem at hand.

If you think you don't have the time to put in such comments, there is yet another type of helpful commenting. You can use blank comments to separate logically distinct sections of code. Inserting blank lines in this way has been shown to be a helpful way to format programs in any language [Ref. 10]. Blank comments are very easy to type! While they don't explain what a program segment does, they at least notify the maintainer that these statements belong together. They should be treated as a unit. Once you've put them in, you may find time to "fill in the blanks" with some explanations later on in the project.

Less is More



Less is more when we are doing diagnosis. The fewer statements and programs we must look at, the better. The purpose of this section is to explore ways to design systems so that we have to look at as few statements in as few places as possible.

It should be obvious that if we could isolate our search to a single function, we would prefer to look at a short function rather than a long one. If we must start looking at the first line, we would prefer that the last line be number 20 rather than number 60. How do we subdivide functions so they are as small as practicable, and yet still meaningful?

One important criterion of whether a function has been decomposed far enough is the sentence test. Can you write a single,
specific sentence which describes the purpose of the function? If you cannot, if you must write a compound sentence with two or more independent clauses with active verbs, you have not reached the lowest level routine. Each of those independent clauses should become a function on its own. The first comment line of each of the functions in the Appendix is a single, specific sentence which describes the purpose of the function.

Another important criterion in determining whether you have broken your problem into fundamental components is the shared function test. Is there any significant segment of the statements in this function which will be repeated in another function? If there is, you have probably not decomposed the problem far enough yet. Put that segment into a separate function which can be shared.

For example, if you know that several of your functions will accept either vector or matrix namelists as arguments, you will probably want to put the process which

normalizes the namelist arguments into a separate function. Such a function might remove leading and trailing blanks and put the names into a matrix with one name left justified in each row.

A third important test in deciding whether your functions have been segmented into their basic parts is the information hiding test [Ref. 9]. Is there a segment of the statements in this function you could split off, which would hide a detail of implementation? If there is, it should probably be a separate function. Several examples of information hiding modules are given below.

1) File cover functions: these hide the details of the file structure from the calling function.

2) Functions which are used as intermediaries to access special data structures.

3) Functions which hide the use of implementation dependent language features.

4) Functions which isolate the details of terminal control for specific hardware.

5) User input functions: these hide the details of how ⍞ is accessed, how buffered input is implemented, etc.

The concept of information hiding modules is an important one. We will explore several such modules in more detail.

File cover functions are an important means of making your applications more maintainable. What they "cover" may be the details of how the particular file system on your application works, or the physical file structure of the application files, or both. A good file cover function will allow you to retrieve and store data by logical identifier, rather than by physical location. Instead of having the application program compute a hash number or look up a pointer in a directory based on an identifier, you can simply pass the identifying number or name to a file cover function. It figures out where the data is, and retrieves it. Because all the details of the file structure are embedded in a few cover functions, it is possible to completely change the file structure with a minimum amount of work. It is even possible to transport the application to a different vendor's implementation, with a completely different file system. The only things you must change are the cover functions. A set of simple cover functions which use a directory scheme to locate data is found in the Appendix.

Terminal hardware control is also a good thing to cover using an information hiding module. Since the hardware vendors generally agree on even less than the software vendors, it is important to isolate their particular requirements from your application program. When you get a new terminal, or your terminal control doesn't seem to be working right, you will have a small number of specialized functions to inspect during diagnosis.

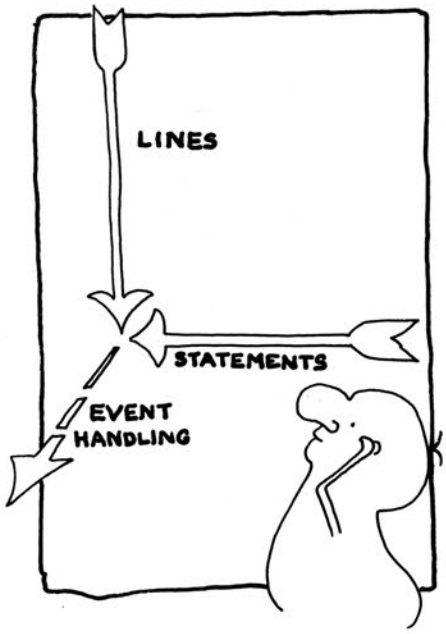
Generalized user input functions are very valuable information hiding modules. If you isolate all terminal data entry problems in a relatively small number of utility functions, you get a number of benefits. Besides the human engineering benefits which are discussed in Lib Gibson's paper on "Designing User-Friendly Systems", such functions

also make maintenance easier. If you need to make a universal change in the behaviour of your dialogues, you only need to look at a couple of functions, instead of hundreds of lines of ⎕ or ⍞ references.

Using information hiding modules is perhaps the most effective strategy for minimizing the number of functions and statements you need to look at when doing maintenance. If you are designing APL applications, you need to become very familiar with this concept.

Since APL was first made available on a commercial timesharing service in 1969, new features have continually been added to the language and operating environment. Two of the most important that have been added to SHARP APL are N-task's and Event Trapping. Other vendors may call them by different names, but their facilities in these areas are essentially the same. In the first case, it is possible to run APL programs in a workspace which is not connected to a terminal. Some implementations, including SHARP APL, allow you to "clone" a copy of an active workspace and split it off to run separately. In the second case, it is possible to "intercept" events (or errors) when they occur. From there, an APL expression may be substituted which will be executed in place of the normal system action. Both of these facilities introduce complications in diagnosing maintenance needs.

You can think of APL multi-tasking and event handling as adding a "third dimension" to an APL program. Statements execute right to left. Branches can take flow of control up and down the page. These features can take the flow of control out into a "third dimension", and then perhaps back into the plane of the original program. The potential for multiplying places to look during maintenance is obvious. Probably the most important guideline we can give you regarding these features is: use them sparingly. If you can do something easily without them, then don't use them at all. In the case of Event Trapping, you need to be especially cautious. The SHARP APL Event Trapping facility may refer to ⎕*TRAP* values which are not directly visible to the program which is executing. Such references can cause tremendous difficulty in diagnosis if they are abused. In general, the safest ⎕*TRAP*'s are those which are strictly local, and those which are strictly global (and not overridden by local definitions). Problems with this type of traps are also the easiest to diagnose.



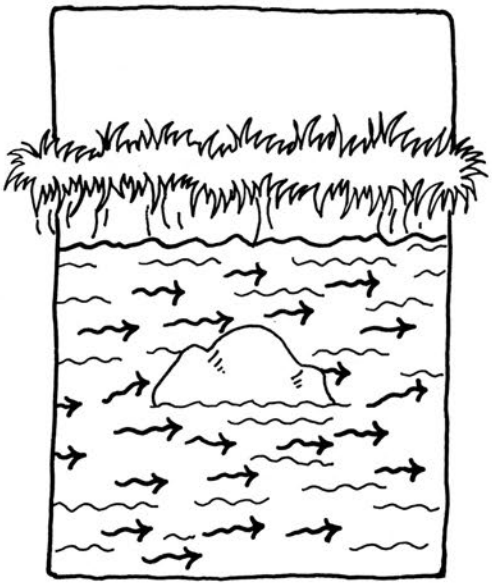
You can design your ⎕*TRAP* definitions with one feature which will make diagnosing problems much easier. Make every trap definition begin with a call to a *TRAPTRACE* function, as the following trap does:

```
⎕TRAP←'▽ 0 E TRAPTRACE ◇ FIXUP ◇ →⎕LC'
```

Most of the time *TRAPTRACE* will have no executable lines. But when you need to check out a □*TRAP*, it becomes invaluable. You can simply have it print the State Indicator Stack and Event Report. You might have it store this same information in a variable for later inspection. You might even have it allow you to enter statements from the terminal to be executed. In this case, it would really be acting more like a programmed stop than a trace.

Using advanced APL features may make the flow of control in your programs nonlinear. This may force you to look in more places when diagnosing a problem or change. When you design applications which use such features, make sure you also design in the means to make diagnosis easier.

Flow of Control



In the world of data processing, "Structured Programming" is a buzzword. It really stands for "Structured Control Flow Programming". (There are other bases to structure programming upon.) In one sense, Structured Programming is of less interest to APL programmers than to those who use traditional programming languages. Much of the looping done in other languages is built into APL's functions and operators. Despite this, there is a lot of branching done in APL programs. Appropriately structuring flow of control is an important way to reduce the number of places you must look to diagnose a maintenance request.

First of all, we want to minimize branching. We want to do this because branching algorithms usually require more statements than those that don't branch. Branching complicates a program, and frequently costs more.

We use branching to do two things: choose a set of statements based on a condition, or repeat a set of statements until a condition occurs. The first type of branching can frequently be avoided by re-coding your algorithm in terms of boolean operations. The second type can be avoided by processing entire arrays simultaneously. In fact, there is really only one good reason to loop in APL. This is when all the data to be processed is not in the workspace at the same time. This can happen in three situations:

1) Someone is entering the data at a terminal.

2) An autonomous cooperating program is providing the data. (An auxiliary processor or batch APL program connected through a shared variable.)

3) The data cannot all be stored in the workspace at the same time. (The data must be stored in a file.)

If your loop doesn't fit into one of these three situations, it can probably be done more directly without looping.

If we really must branch, how can we do it so as to minimize the effort required to diagnose a change? We want to make our branching patterns as similar as possible to sequential flow control as possible. This leads us to three guidelines for "safe branching":

1) The line number of the target of a branch statement should be higher than the line number of the branch statement itself. In other words, branches should go down the page. The only exception is branches which are returning to the top of a loop.

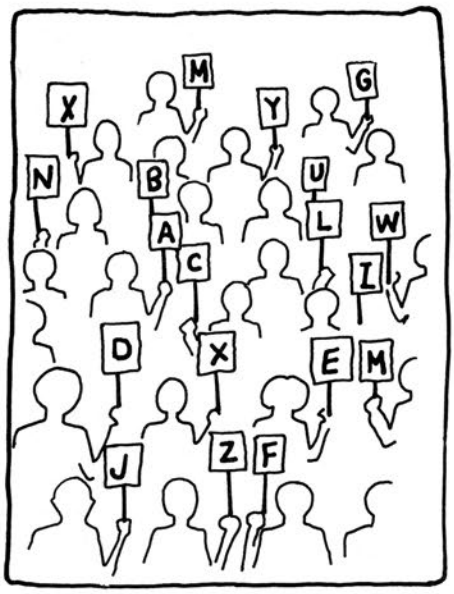
2) The segments of a program which are the targets of a branch statement should be contiguous to each other, and one segment should be contiguous to the branch statement itself. In other words, conditional branch statements should be kept as close as possible to the statements they control.

3) Loop limits should be tested at the top of the loop. Loop counters should be incremented at the top of a loop.

The effect of the first two rules is to keep flow of control as close to sequential (top to bottom) as possible. The motivation for the first part of the third rule is make it possible for a loop to "do nothing" gracefully. If a loop is tested at the bottom, it will always execute at least once, even if there is nothing to do! The motivation for the second part is to prevent problems if there is more than one statement which returns to the top of the loop. If the counter is incremented at the top, there can be many separate returns to the top of the loop without problem. If the counter is incremented right before a branch to the top of the loop, there must be a separate incrementing statement *for each returning statement*. This increases the number of statements which must be written, and which must be inspected during diagnosis. Leaving out one of the incrementing statements is a very effective way of generating an endless loop.

In most other programming languages, flow of control statements are necessary in any non-trivial program. In APL, we can frequently avoid branching altogether. If you want to design maintainable systems, you will seek to keep your use of branching as simple as possible.

Naming Conventions



Every APL programming standard I have seen has had some suggestions about naming conventions. Unfortunately, there is very little agreement among them. Therefore, before we suggest a naming convention, we should look at just what you can do with a name.

What kind of information can a name convey? First, it can convey the object type (function, variable, label, or group) of the name's referent. This is helpful, but since the information is easily and inexpensively obtained by using *□NC*, it isn't really necessary.

Second, if the object is a variable, objects of different rank can be distinguished by name. This is in fact what was done in *A Programming Language* [Ref. 4]. In that book, the names of scalars, vectors, and matrices were printed in different type fonts. Once again, however, the information is easily obtainable by other means (*ρρX*). In addition, a variable may change its rank, in which case the convention would lead you astray.

Third, if the object is a variable, the data type may be distinguished. Here too, however, there are simpler ways to determine whether a variable is character or numeric (`0=1↑0ρX` or `~∨/□AV∈X`).

Fourth, the scope of the object may be distinguished by a naming convention. Functions and variables can be global in the workspace, global with respect to some functions but localized in one, or strictly local. There is no easy way to determine the scope of an APL object. You can write a program to analyze all uses of the name in all functions in the workspace if you need to. It will not be inexpensive to run.

Fifth, the use or purpose of an object can be distinguished by a naming convention. An object might be a loop head label, a loop counter, an index constant, etc. Once again, it's possible to determine this easily. Some of these type of uses could be identified by a workspace analysis program.

Finally, the meaning of the object in the application context can be indicated. There is no automatic way to determine this most important fact. This is where the semantics of the application become clear to the programmer. *A-B* can mean a lot of things, but *ACTUALS-BUDGET* narrows the choices considerably.

There are several ways you can construct APL object names to indicate extra information. These involve the use of prefixes, suffixes, and reserved letters. You might choose to indicate those variables which are global by prefixing their names with *GΔ*. You could distinguish all labels which stand at the head of a loop by suffixing the label name with the letters *LOOP*. You might decide to restrict the use of the Δ, ⍙, and the underscored alphabetics to the names of certain kinds of objects only.

There are also some special problems to consider when designing a naming convention. Are you going to allow the use of numeric digits in names? Will they have any special meaning? Will you encourage, forbid, or regulate the use of single letter names? What about acronyms, abbreviations, and names containing more than one word? Should labels be given special names? And the question of the century, so aptly phrased by Jon McGrew, "DO THR CHR NMS RUN FST?" [Ref. 7].

Since the information regarding object type, rank, and data type is so easily obtained by other means, we will exclude these from further consideration. Clearly, if a naming convention conveys the scope, use in the program, and meaning in the application context of a name, it is providing a lot of useful information. That extra information makes the task of diagnosis a lot easier.

The special problems can be quite thought provoking. Why do people use numeric digits in names? Some use them to create names which are unlikely to cause name conflicts with the global environment. This is necessary only very occasionally. Others use them to distinguish otherwise indistinguishable objects. This is never necessary. If you can't associate a simple word with your variable, I question whether you really know what it represents. Once in a while, the nature of the application suggests a name which contains a numeric digit. If your system processes transaction types 1, 2, and 3, you might have a function named *PROCESSTRANSACTION2*. In general, however, you should avoid the use of numeric digits in names.

Why do people use single letter names? Well, you can't make them any shorter! They're easy to type. Some APL systems store short names (3 or 4 letters or less) in a different way than longer names. This means, in the case of SHARP APL, that the name *ABC* takes a symbol table entry only. In contrast, the name *ABCD* takes a symbol table entry, plus 16 bytes to store the name.

Why shouldn't you use single letter names? The answer is found in the concept of psychological "distance" between words [Ref. 11]. Distance is the number of positions in which the words differ. The words "duck" and "dock" have a distance of 1. The words "duck" and "dice" have a distance of 2. In general, the greater the distance between two words, the less likely it is that they will be confused.

There are two other factors involved in distinguishing names. The first and last letters of a word are more significant in making distinctions. You are less likely to confuse "duck" and "muck" than "duck" and "dock".

In addition, some letters look more alike than others. If you are using SHARP APL, you might find the following experiment interesting. Try typing the following pairs of characters on top of each other. See which overstrikes look so similar to some other character that APL accepts them as valid.

CG EF IJ IT OQ PF RP XK

The lesson from each of these perspectives is that if you want to design maintainable systems, you should choose names which are not easily confused in reading or writing.

If you use single letter names, which have a distance of 1, it is very easy to misread one name for another. This will slow down diagnosis. If you limit yourself to single

letter names, a single keying error will very likely yield another legitimate name. If you've never had to track down an error that was due to someone typing *I* instead of *J*, I can assure you that it is no fun.

Why do people use acronyms? Well, it's a lot easier to type "FBI" than it is to type "Federal Bureau of Investigation". The problem with acronyms is knowing the context in which to interpret them. Do you know all the other meanings of "APL" in public use in North America today? There are a number! If the maintainer can't figure out what your acronym means, diagnosis will take longer. This problem can be overcome if you document the meaning of the acronyms.

In general, abbreviations have the same problems as acronyms. If you abbreviate anything enough, you'll get gibberish. The problem is balancing the effort required to type a name against the information it conveys to the reader. In addition, you can abbreviate by truncating or by condensing (*ACC* or *ACNT* for *ACCOUNT*). Condensed names usually convey more meaning.

Using special names for line labels is a very common habit among APL programmers. Obviously, □*NC* can be used to determine that a name refers to a label. Why, then, is this habit so popular? The main reason seems to be that by making label names stand out, the branch destination is easier to find when reading a program.

There are several approaches to naming labels, which can be called the FORTRAN approach, the ALGOL approach, and the Context approach. The FORTRAN approach chooses labels like *L*10, *L*25, *L*50. The number of symbol table entries is minimized, but these can be very confusing if they don't appear in ascending order in the function text. The ALGOL approach chooses names like *IF*, *THEN*, *LOOPA*, *ENDA*, etc. These impart the purpose of the label in a control structure, but can be easy to mistype if suffixes are needed to distinguish them (*IF*1 versus *IF*2). The Context approach chooses the name based upon the purpose of the statement which is labelled: *ERROR*, *CHECK*, *ACCOUNTLOOP*. More symbol table entries may be used, but more information is conveyed. Each approach has advantages and disadvantages.

As for "DO THR CHR NMS RUN FST", the answer is No! They do take a little less space. They take an infinitesimal amount less of CPU time to scan when they are first typed in, and when the function is displayed. Every serious APL implementation, however, replaces the names in a function with some sort of pointer. Therefore, 3-character names or 30-character names have no effect on *execution* speed.

What naming conventions do I recommend? One standard almost everyone will agree to is that functions should have meaningful names. After that, there is no consensus. The rules listed below have been used successfully by a number of people I have worked with. They aren't the only possible choices. They do address all of the questions explored. They also tend to minimize the effort required to understand a program and diagnose a change.

1) Choose names which are meaningful in the application context.

2) Choose names which suggest their use in the program.

3) Don't use names which have a distance of 1 from another name used in the same program.

4) Don't use numeric digits in names.

5) Don't use acronyms unless you provide a glossary for the programmer.

6) Avoid the use of abbreviations by using common (short) words.

7) When you do abbreviate, condense names rather than truncating them.

8) Don't use any underscored alphabetics in the names of variables which are strictly local.

9) Underscore only the first letter of names which are global to some functions but are local to others.

10) Underscore all the letters of names of objects which are global in the workspace.

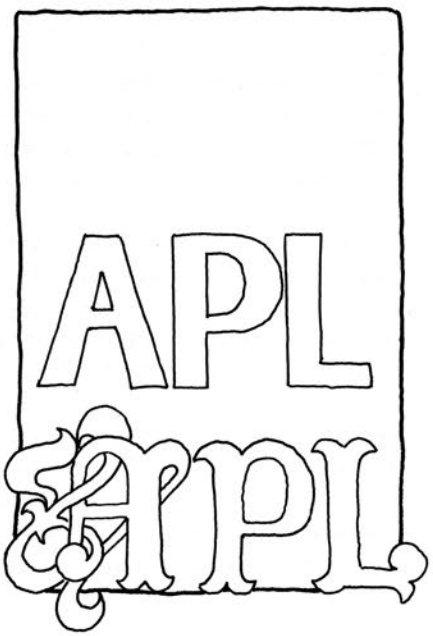
11) Use a consistent label naming convention.

You may wish to design your own naming convention. If you consistently address the issues discussed above, it will be a great help to you in reading and diagnosing maintenance problems.

Statement Style

By statement style, we mean all the little things that go into making a single statement transparent or opaque. The following issues are hotly debated:

1) Embedded assignment
2) Statement gluing
3) Nesting of parentheses and brackets
4) Statement length



An embedded assigment is one which is not the last operation carried out in a statement. We can dismiss the assignment of the same value to several variables (*A*←*B*←*C*←3) as a non-issue. The interpretation is unambiguous. The problems arise when you assign different values to the same variable (*A*←(*A*←5)+*A*←3) or different values to different variables (*A*←3+*B*←2). We'll start this discussion with an historical note. In

1973, the co-developers of APL at IBM, Ken Iverson and Adin Falkoff, wrote the following,"It is interesting to note that the use of embedded assignment was first suggested during the course of the implementation when it was realized that special steps were needed to prevent it" [Ref. 2]. Some of us wish they had taken those steps.

There are two reasons why you shouldn't use embedded assignments. The first is that the effects of embedded assignments vary from implementation to implementation. In other words, your program may give one answer running on one vendor's software, and something quite different somewhere else. In fact, you may get a different answer from different versions of the same vendor's software! A fundamental rule of programming is that you *never, never* use a language construct in a way that you know will give a different answer if you use someone else's version of the programming language. That rule is good advice for FORTRAN programmers, COBOL programmers, and even APL programmmers.

Well, you say, the behaviour might get standardized someday. Maybe it will. But if you are concerned with writing maintainable software, there is a far more important reason for not using embedded assignment. Embedded assignment is the APL equivalent of run-on sentences in English. An assignment completes a thought. When you put several complete thoughts into an English sentence, you get a run-on sentence. When you put several thoughts into a single APL statement, you get a one-liner. Both are unnecessarily difficult to understand. They increase the time required to understand a sentence, or APL statement, for no good reason.

To prove the point, listed below are a couple of statements taken from applications running on the SHARP APL system. As you can see, statements with embedded assignments are as hard to understand as run-on sentences in English.

```
Z←(W,(+/R),+/C)⍴(R←0⌈⌈/⌈⍀AρR LEV C←0⌈⌈⍀⌈⍀AρC)SH(A←(×/W←¯2↓A),¯1↑A←⍴W)⍴FM
W←M W LEV R←C←''
```

```
Z←((⍳¯1↑T)∈T←+\1+¯1↓0,+⍀0=T∘.|⍳A)⌿((A←¯1↑T←×\1↓T),1↑T←⌽⍴W)⍴W←M W
```

Statement gluing is the practice of taking complete APL statements and forcing them be on the same line by connecting them with constructs like ,0⍴. People who are using implementations which provide a diamond statement connector (◇) have no excuse for using this sort of thing. If you simply must put several statements on line, use diamonds. If you don't have the diamond available to you, you might think that this practice is justified. On what grounds? If you are using SHARP APL, you will save 6 bytes for each statement you cram together with another by using ,0⍴. If you need to save 6 bytes, either you are using a micro-computer, or you have not designed your application properly. If you are using a micro, you have my sympathy. I remember the 30K workspaces of APL\360. If your problem is a poor system design, you should treat the cause, not the symptoms.

Once again, a few ugly examples are worth more than a thousand sermons. Again, these come from commercial APL applications.

```
→((ρY)≥L←L+1+0×L←L+1)ρL30,0ρρAC←0×AC+0×DC←DC+AC+0×ρρT←0×T

→(3≥M←M+1)ρL20,0ρρDC←0×DC+0×CC←CC+DC+0×ρρDT←0×DT
```

Statement gluing also has some undesirable side effects, apart from making statements more difficult to read and diagnose. It slows down execution on some systems. It makes the program trace debugging feature less useful, since trace only shows the last array computed on a line. (The same is true of using embedded assignment.) It tends to make statements unrestartable. It has a side effect of turning scalars into one element vectors (as in `X←3,0ρY←ι5`). Don't use it.

There is experimental proof that people have difficulty handling nesting caused by parentheses or brackets when the nesting level is greater than 3 [Ref. 11]. If you are trying to design a maintainable system, you won't want to write statements which are difficult for people to understand. If you have a statement in which you have nested brackets or parentheses 4 or 5 deep, you should think about breaking it into several statements.

It wasn't hard to find examples of over-nested statements. Can you say what nesting level the character in the middle of each statement is at, without counting in from either end?

```
HIP←DIR[(DIR←DIR[ι2;]),[1] (DIR[DIR[;1]ιHV2;],[1] 2 0↓DIR)[;1]ι22;3]

→(0=1↑ρPOM←POM[J←((ABϵ1 2 5)∧~((10×DIR[N2;1])+PH)ϵNBNC)/ι1↑ρPOM
      ←POM[J;6+ι12];])ρL10
```

Have you ever had to maintain a system which contained hundreds of statements like those listed above? If so, you know how frustrating it was. The only way to stamp out such nonsense is to follow the Programmer's Golden Rule.

Every APL programming standard I have seen has attempted to legislate some maximum line length. These lengths always seem to be based on the width of the terminal the author is used to working on. The motivation is correct, but the measure is bad. The complexity of an APL statement is based on the number of symbols and names, not the number of characters. If you don't use embedded assignments, statement gluing, or statements with a nesting level greater than 3, you will rarely write a statement which is too complicated for an average APL programmer to understand.

The specifics of this section may be subject to heated dispute. There is no dispute about the fact that if you make APL statements needlessly complex, you will slow down the process of diagnosis. That will cost someone time and money.

Red Herrings

One of the most frustrating experiences in diagnosing is an investigation which leads to a dead end. Either you didn't need to look at the program at all, or you are looking in the wrong place entirely. One of the chief causes of such fruitless searches are program red herrings. A red herring is something that distracts from the matter at hand. Some of the most common APL red herrings are listed below:

1) Comments that don't correspond to the code.

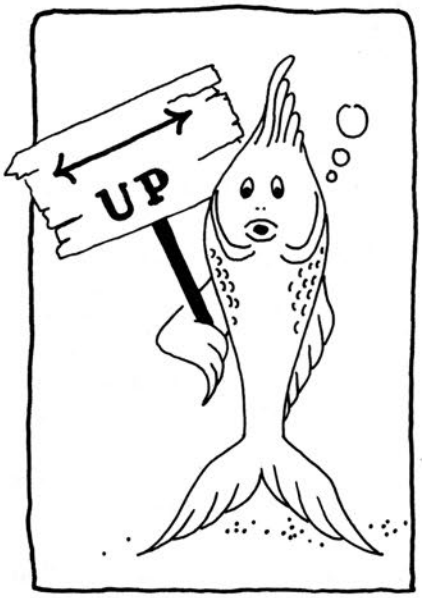
2) Names that are localized and not used.

3) Variables that aren't localized but aren't used for communication between functions.

4) More than 1 function (in a file containing functions) which has the same name but does different work.

5) Functions in a workspace which are never used.

6) Superfluous generality.



If you aren't going to keep comments up to date, don't put them in. If you can't resist commenting (most people can), you might want to put your comments in the past tense (a suggestion from Tom Cooper of Xerox).

```
⍝ THIS SECTION USED TO COMPUTE THE FRAMISTAT FACTOR
```

When I find localized names that aren't used, or unlocalized names that ought to be, I feel like I'm picking up someone else's dirty laundry. It's evidence that the previous maintainer was in a hurry, being careless, or indifferent to the quality of his work.

A diagnostic cross-reference program will happily flag such occurrences. If you don't have such a program available to you, you can at least catch names that ought to be localized in the following way.

```
BEFORE←⎕NL 2
FOO
AFTER←⎕NL 2
⎕←(~∨/AFTER∧.=⍉BEFORE)⌿AFTER
```

Functions lying around that are never used are more dirty laundry. A workspace cross reference listing can help you find them.

Generality is a quality we normally want in programs. But sometimes people make code more general than will ever be required. There are two kinds of superfluous generality. With the first kind, we find people who use a more generalized primitive, when a more specialized one would have worked as well. This is sometimes the result of trying to be very concise. The examples below show some relatively harmless expressions of this type.

More General	**More Specific**
`+/2↑VECTOR`	`VECTOR[1]+VECTOR[2]`
`VECTOR∈SCALAR`	`VECTOR=SCALAR`
`100⊤VECTOR`	`100|VECTOR`
`2 1⍉MATRIX`	`⍉MATRIX`
`SCALAR∘.=SCALAR`	`SCALAR=SCALAR`

Even these examples leave us scratching our heads in curiosity.

In the first case, we have to wonder if the 2↑ was just the easiest way to get the first two elements. Or, did the programmer know that sometimes the array would have 0 or 1 elements? In that case, he was programming defensively to avoid an *INDEX ERROR*. In either case, unfortunately, if we are conscientious maintenance programmers, we must search and find out if this fragment is just adding two numbers. That will cost us time. If the vector will always have at least two elements then this may be a red herring.

The second kind of superfluous generality occurs when someone writes code which is almost generalized. The program segment below is an example.

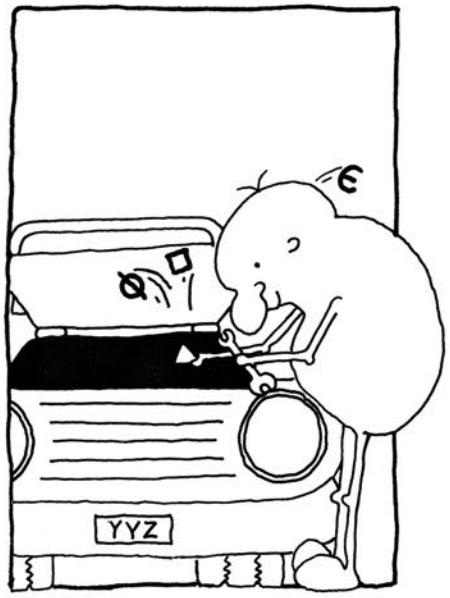
```
[1]  RESULT←0 12⍴0
[2]  CTR←0
[3]  LMT←1↑⍴MATRIX
[4]  LOOP:CTR←CTR+1
[5]  →(LMT<CTR)⍴END
[6]  RESULT←RESULT,[⎕IO] PROCESS MATRIX[CTR;]
[7]  →LOOP
[8]  END:
```

On line [6], we see an Index Origin generalization. It's nice to have origin independent software, but this isn't. If anyone executes it in origin 0, the fact that the loop counter and limit are based on origin 1 will cause it to fail. Don't make your programs look more general than they really are. Don't force the poor maintenance programmer to go off on a wild goose chase examining circumstances under which the program will never run.

Diagnosis is a process of searching and identifying. Providing documentation, using modular functions, and using disciplined flow of control make it easier to locate that portion of the system which must be changed. Following a consistent naming convention, writing in a clear statement style, and avoiding red herrings make it easier for you to determine exactly what to change. If you follow these recommendations, you won't need the sleuthing skills of a Sherlock Holmes to find what you're looking for.

MODIFICATION

Once we have decided what to change, we want to make those changes as easily as possible. We do this by reducing the quantity and increasing the quality of the statements and values to be changed.



How can we reduce the quantity of APL statements and data values which must be changed? There are several general methods:

1) Write systems which are parameter-controlled (table-driven).
2) Write systems which take advantage of APL's delayed binding time.
3) Write systems which are built of interchangeable parts.
4) Do everything you can to make life easier for the maintainer.

These techniques can significantly reduce the time you spend changing programs.

Parameter Control

The word "parameter" is currently being trivialized by government bureaucrats and other verbose personages. It is a useful word, however, and maybe we can take it back from them. The American Heritage Dictionary defines it as, "A variable or an arbitrary constant appearing in a mathematical expression, each value of which restricts or determines the specific form of the expression". In other words, it is information that controls a procedure. One way we can make modifying APL applications easier is by designing them to be controlled by sets of information that have two properties:

1) The sets must be abstracted from the procedure.
2) The sets must be easily changed.

People have in the past referred to applications designed in this way as "table-driven systems". Perhaps this designation came from COBOL, where arrays are called tables. People normally equate matrices and tables. But since the "tables" which control the application may be vectors or cubes, "parameter controlled" is probably more accurate than "table driven".

Let's begin this section by looking at the concept of a "symbolic constant". Many applications make repeated use of a few constant values. These arrays appear as literal or numeric constants embedded in programs.

In a word processing application, you might find constant vectors like the following in your programs.

```
'AEIOUY'
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
'.:,;!?-'
```

In the dynamic world of APL applications, allowing such values to be equated conceptually with the class of information you think they represent can cause you problems. In other words, constants change.

If you have "hard-wired" these values into your programs, and then discover later that you need to add or delete an element, you will have a large editing job ahead of you. If you had assigned them to a global variable, the change is trivial. Kernighan and Plauger refer to such values as "symbolic constants" [Ref. 5].

The following lines come from an APL program currently in use on the SHARP APL system:

```
G←(2 2⍴16.86 5.74 1.92 0.531)[1+(I[1]=0)∧(P=1)∧1=⍴T;2-D]

G←(G×(0.23×H)+(0.11×~G))+((0.38×H)+0.17×~H←MP[7;]≥50)×~G←0>-⌿MP[4 3;]

U←,(R-P←⍉R←(G,G)⍴1+⍳G←¯1+S⌈T)∈¯1 ¯3 ¯5

A←500×(1÷,Q+.×⍉U∘.*0 1 2 3 .5 .3)-0.000192
```

I suspect that some of the interesting constants should be replaced by symbolic constants. Unfortunately, the programs were totally undocumented, and consequently, no one really knows for sure!

What is a legitimate candidate for a symbolic constant? We can start by ruling out a few possibilities. The most commonly occurring constants in APL programs are the scalars ' ', 1, 0, ¯1. They rarely represent an underlying class of values. In addition, constants used in indexing can be excluded. (See the section *Binding Time* for a discussion of index constants.) Scalars not in these categories may represent a single instance of a class of values that should be stored in a symbolic constant. The most important group of such constants are vectors which occur in several programs. If the operations which are being performed on these constants are comparisons (= ≠ ⍳ ∈), then they are even more likely to represent a set of values which may be changed.

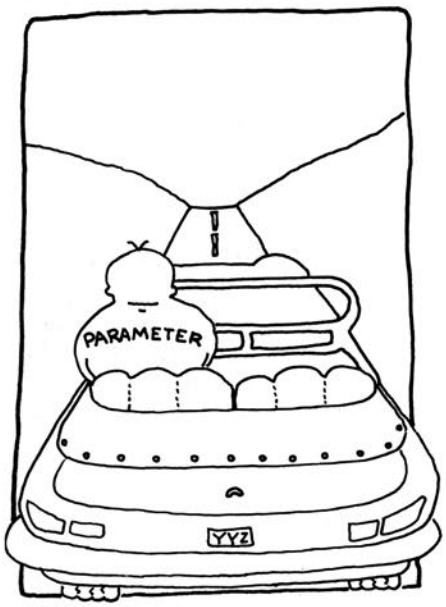
If you have put a commonly used constant into a global variable, you have abstracted it from the procedure. You have also made it much easier to change, at least for a programmer. Whether you should provide facilities for an end user to change it is another question. In many applications, the test is a simple one.

Can you conceive of a situation in which someone will want to change the elements of this parameter?

If the answer is yes, you should provide an interactive editing capability which can be used by a non-programmer. This can significantly reduce the amount of time you spend maintaining an application. The user can do his own maintenance! Such an

editing facility implies storing the parameter on a file, since most APL systems do not allow dynamic execution of)*SAVE*.

The Appendix lists a program (*EDIT*Δ*TABLE*) which allows such interactive editing of vectors, matrices, etc. It is meant for use with tables which are essentially independent of one another. If two tables are related in some way, it is best to write a special program to edit them at the same time. Such a dependency would exist between a table of department names and department numbers. There should be as many names as numbers. You shouldn't count on the user to always change one if he changes the other. Therefore, the two tables should be edited together.



If you really can't think of a situation in which someone would want to change this parameter, put it into a global variable. You'll probably have to change it anyway, but the changes should be relatively infrequent.

We have now discussed how to identify application parameters and how to change them. The last question to consider is how to use them.

Your programs should assume that parameters will change. Not only will values change, but shapes as well. Programs should not assume that tables will have a specific shape. At least one dimension of a table will vary. Some rules for handling such variations are given below.

1) Calculate the number of times a loop will be repeated based on the shape of the parameter.

   Instead of writing:

```
LOOP:→(57<CTR←CTR+1)⍴END
```

   use

```
LMT←1↑⍴TABLE
LOOP:CTR←CTR+1
→(LMT<CTR)⍴END
```

2) If a dimension of a parameter may take any length, let it grow naturally through use.

```
NEWNAME←get new name
WIDTH←(¯1↑⍴DEPTNAMES)⌈⍴NEWNAME
DEPTNAMES←(((1↑⍴DEPTNAMES),WIDTH)↑DEPTNAMES),[1] WIDTH↑NEWNAME
```

3) If you must force a specific width for display purposes, do it at the time the display is prepared.

   Instead of forcing a table to have 25 columns and printing it like this:

```
'I10,X4,25A1' ⎕FMT (DEPTNUMS;DEPTNAMES)
```

   fix the shape for printing purposes only like this:

```
'I10,X4,25A1' ⎕FMT (DEPTNUMS;((1↑⍴DEPTNAMES),25)↑DEPTNAMES)
```

We could make other suggestions on using tables like this. The main point is that once you have generalized an application by making it parameter controlled, you should write all code which deals with that parameter in as general a way as possible. In this way you will reap the maximum benefit from the abstraction you have introduced.

Binding Time

A second way to minimize the effort required to make changes is to make use of APL's "binding time". This is the term computer scientists use to describe when a name is associated with the object it refers to. APL has late binding. This means that the APL interpreter waits until the very last moment to decide what a name refers to.

Let's look at an example. The APL statement listed below has several valid interpretations.

```
F G H
```

1) *F* and *H* are variables; *G* is a dyadic function.

2) *H* is a variable; *G* is a dyadic function; *F* is a niladic, explicit result function.

3) *H* is a niladic, explicit result function; *G* is a dyadic function; *F* is a variable.

4) *F* and *H* are niladic, explicit result functions; *G* is a dyadic function.

5) *F* is a monadic function; *G* is a monadic, explicit result function; *H* is a variable.

6) *F* is a monadic function; *G* is a monadic, explicit result function; *H* is a niladic, explicit result function.

We haven't even begun to consider each of the different ranks and data types that the variables could take on!

Other languages handle this by forcing you to say in advance what every symbol means, before you use it. If we spoke like this in English, our conversations would sound like the following:

Let "Erika" signify a proper noun.
Let "saw" signify a transitive verb.
Let "the" signify a definite article.
Let "house" signify a noun.
"Erika saw the house."

Now you know why FORTRAN programs are so much longer than APL programs! In APL, like natural language, the nature of a symbol is determined by its usage in context. (For example, "saw" can be a noun, and "house" a verb.)

Most APL programmers do not take full advantage of APL's late binding. As a result, their programs are not as maintainable as they might be. Let's see why.

The two functions listed below do the same thing:

```
    ∇ ASK1
[1]   ⍞←PROMPT,' '
[2]   RESPONSE←(1+⍴PROMPT)↓⍞ ∇

    ∇ RESPONSE←ASK2 PROMPT
[1]   ⍞←PROMPT,' '
[2]   RESPONSE←(1+⍴PROMPT)↓⍞ ∇
```

The difference is that the first uses global variables to communicate with calling functions, and the second uses an argument and a result.

The problem with the first program is that the nature of the interface is bound at the time it is written. In contrast, the nature of the second program's interface isn't bound until it is executed.

Every programmer who wants to use *ASK*1 must know that *PROMPT* is the name of its input, and *RESPONSE* is the name of its output. Not only that, he must know this fact *at the time he writes his own program*. That is when the interface is bound.

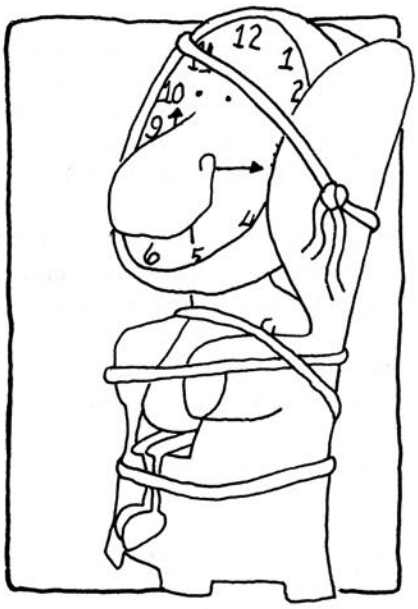
In contrast, the programmer who wants to use *ASK*2 does not need to know what the names of the argument and result are. He is not bound to those names. He can call the function with different names.

```
ZZ←ASK2 XX
```

He can even call the function without any names at all.

```
⍴ASK2 'QUESTION:'
```

The values that *RESPONSE* and *PROMPT* have are not bound to those names until the moment when *ASK*2 is executed.



This is an obviously simple case. But what if *ASK*1 were called by 50 different functions? If you decided that you absolutely had to change the name of the input variable, you would have to change 50 functions! Changing program interfaces happens regularly during maintenance. The way to make it easy is to use arguments and results for interfaces whenever possible.

Not only are names bound to objects, but rank and shape are bound to variables. Programmers who use traditional programming languages are accustomed to DIMENSION statements, DECLAREing variables, etc. These conventions tell the language processor how to allocate memory for the program's variables. We don't have to do this in APL, but some people do it implicitly without realizing it.

When you write *R*[;3]+*B*, you have in effect declared that *R* is a matrix with at least 3 columns. If you had said *R*[;*A*]+*B* you have still implicitly declared that *R* is a matrix. You have, however, postponed the determination of the shape until the statement is executed. You have thus taken advantage of APL's late binding time.

You should be wondering how all of this enhances the maintainability of an APL application. Let's look at some examples.

Assume that you have a file directory stored in a character matrix. The first 4 columns are the literal representation of the file component number. The remaining columns identify the data by name. If you have used numeric constants for indices, and you decide that you absolutely must add another column for the component numbers, you're in trouble. Expressions like the following won't work.

```
⍎DIR[ROW;1 2 3 4]
4↓DIR[ROW;]
⍎,' ',((1↑⍴DIR),4)↑DIR
(0 4↓DIR)∧.=ID
```

You are going to spend a lot of time changing your program in this case.

If you had stored the *indices* of the these fields in *index constants*, however, the change would have been trivial. All you would need do is change the values of 2 global variables. We might call them *ICOMP* and *INAME*. Then the expressions listed below would have worked before and after the change.

```
⍎DIR[ROW;ICOMP]
(⍴ICOMP)↓DIR[ROW;]
⍎,' ',((1↑⍴DIR),⍴ICOMP)↑DIR
DIR[;INAME]∧.=ID
```

The original value of *ICOMP* would have been 1 2 3 4. When you changed the data

structure, you would have assigned 1 2 3 4 5 to it. *INAME* would be similarly changed.

As a second example, assume that you have a data base of accounting information. Each record contains the following fields:

Division, Department, Account, Subaccount, Amount, Check Number

In your file, you store these records blocked together in numeric matrices with one column for each field. You will find lots of statement fragments like the following in your system:

```
+/RECORDS[;5]
ACCOUNT=RECORDS[;3]
```

You might even find some attempts at generalizing the indexing:

```
TOTAL←+/RECORDS[;¯1+¯1↑⍴RECORDS]
```

Unfortunately, changes to the data structure can make all of these wrong. If you must delete the Division field, and insert 2 columns for Date and Description Code after the Amount field, you've got a lot of work ahead of you.

If you had set the following index constants, you would merely have had to change their values, and add two new ones. The original values would have been the following:

```
IDIV←1 ◇ IDEPT←2 ◇ IACCT←3 ◇ ISUB←4 ◇ IAMT←5 ◇ ICKNUM←6
```

The program fragments would have looked like the following:

```
+/RECORDS[;IAMT]
ACCOUNT=RECORDS[;IACCT]
```

Interestingly, this technique has two beneficial side effects if you are using SHARP APL:

1) The programs run slightly faster. This is because the constants do not have be set up each time they are used.

2) The programs take somewhat less space. This is because each constant has a storage overhead in the function which the variable does not. If you use the same index many times, the entry used in the symbol table, and the space consumed storing the scalar index value as a variable, will be amply repaid.

Obviously, the use of index constants also improves the readability of a program. You don't have to figure out what the reference to `A[17;]` means. The name of the index constant tells you that. This will reduce diagnosis time as well.

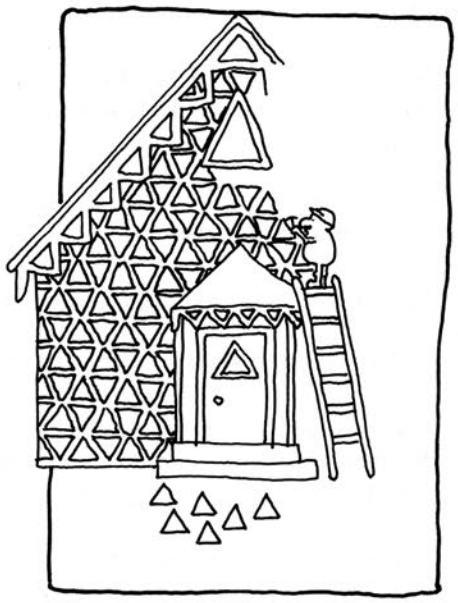
Index constants are not a panacea. You don't need to use them everywhere. They may be unnecessary for a data structure known only by a single program. But when you

have an important data structure, known by several programs, you can use them to delay binding time and enhance maintainability.

Interchangeable Parts

When Henry Ford revolutionized the automobile industry, he did so based upon several important principles:

1) Mass production
2) Division of labour
3) The assembly line
4) Interchangeable parts



He wanted to minimize the effort required to produce a car. We want to minimize the effort required to modify a program. We will therefore investigate the one principle which is related to maintenance as well as initial assembly—interchangeable parts.

The basic idea behind interchangeable parts is that because parts have qualities of uniformity, if a part fails, you can simply remove it and substitute another. The next time your APL terminal breaks down, watch the service representative closely. He won't test individual transistors, capacitors, etc. He will pull entire boards and assemblies and replace them. He can do that because each distinct board and assembly is the same size and has the same connections. Such boards have been built to be replaced.

Ephraim McLean introduced the notion of "Throwaway Code" to the APL world [Ref. 8]. He suggested six rules for producing such programs:

1) No APL function should ever be longer than one page.

2) All APL functions should have but one starting and one termination point.

3) All variables within an APL function should be defined as local variables.

4) All APL functions should be defined with explicit results.

5) All APL functions should contain sufficient documentation so that the entire function can be recreated without reference to the code itself.

6) Do not attempt to rework APL functions; when the need arises for modifications, throw the old code away and rewrite from scratch.

His aim was to encourage a style of APL programming which would make APL a maintainable language in a production environment. By following the philosophy of interchangeable parts, he felt that two other benefits would also accrue:

1) Enhancements which have been introduced to the APL language since the program was written will be incorporated into the new version.

2) Many people have a low view of program maintenance. This view would be changed as people recognized that the maintainer does as much creative work as the originator.

The philosophy that McLean espouses is basically a good one. It needs several improvements to be viable. Let's go back to the interchangeable parts analogy.

To be interchangeable, parts must have sufficiently small value that they can be discarded without economic hardship. McLean's Rule 1 is too broad to ensure this. In general, a page of code costs too much to throw away with impunity.

In studying APL applications which I considered to be well modularized, I found that the *average* number of statements per function was between 12 and 15. You don't suffer greatly when you junk functions of this size. Moreover, when you state the size goal in terms of an average, you provide flexibility for the odd case where you really need a larger function.

A more precise measure of the value of a function should be based on a quantitative complexity measure. Halstead's "Software Science" is a widely recognized method for measuring software complexity [Ref. 3]. The relationship between APL and "Software Science" is discussed in DeKerf and Mauri [Refs. 1 and 6]. The metric "program volume" and the related metric "effort" could be used to determine whether a program is too valuable to be scrapped.

To be interchangeable, parts must have the same connections. These connections must be standardized, even between manufacturers. The leads and sockets of a printed circuit board correspond to the inputs and outputs of a function. McLean's rules 3 and 4 suggest that all functions should communicate through arguments passed and results returned. This is an essential minimum. Some further requirements can be defined:

1) Provide the sort of "control knob" arguments which will make your function more generally applicable. (Manufacturers try to build certain types of assemblies sufficiently general that they can be used in different products.)

2) Follow APL's convention and pass these control inputs through the left argument (like ↑ ↓ ⌽ ⍉).

3) If you are using an APL implementation like SHARP APL which allows ambivalent functions, provide convenient default values when the left argument is omitted (like - ÷ ⋆ ⍟ ⍉).

4) When an argument represents a set of choices, make it easy to specify "all elements of the set". If possible, follow APL's example and let an empty specification mean all values (like *M*[*I*;]).

5) When you think you have more than two arguments, split the function into several functions until you have two per function. If you follow this strategy, and use the concepts of application parameters and index constants introduced earlier, you can almost always use just arguments and result for functional input and output. If you are using an APL implementation like SHARP APL, which has enclosed arrays, there is never a situation in which you need to use global variables for communication between functions. You can always enclose several distinct arguments, catenate them together, and pass them as a single argument.

If you follow these guidelines, you will have functions which are *easy* to connect to others.

To be interchangeable, parts must do the same thing. McLean's rule 5 on documentation is concerned with this problem. If we are going to throw away a program, we must be able to re-create its former function. This means that internal documentation should emphasize *what* a function does, and not be so concerned with *how* it does it, since the *how* may change entirely. Such documentation can be placed in a block of comments at the beginning of the function. The following topics should be covered:

1) Purpose: a single sentence should describe what the function is used for.

2) Inputs

   a) Right and Left Arguments: give the rank, shape, data type, and meaning of each of the arguments.

   b) Files: list the names of files read from and a description of the information which is read.

   c) Parameter constants and Index constants used: list their names.

   d) Global variables used as pseudo-arguments: list their names.

3) Outputs

   a) Result: give the rank, shape, data type, and meaning of the result.

   b) Files: list the names of files written to and a description of the information which is written.

   c) Global variables used as pseudo-results: same as explicit result.

   d) Effects: changes to the WS environment, functions fixed or erased, etc.

If you look at the functions in the Appendix, you will see that the comments they contain have been written in the manner described above.

Documentation is not enough to ensure that parts are interchangeable. You must test to make sure that they function the same. The most important test is to pit the original

function against the new, improved version. If they really do the same thing, they ought to give the same result for all classes of valid and invalid inputs. If the two functions work the same under testing, you can have confidence that they are interchangeable.

The question you should be asking is, "Does he expect me to throw out a function every time I make some minor change?" Unlike McLean, I do not espouse junking a function every time it is changed. His rule 6 is too general and somewhat naive. I believe that minor changes should be made to existing functions. I also believe that there is a point at which a program has been changed so much that it is easier to scrap it than to modify it. Remember, we are trying to minimize the cost of modification.

What is the "trashcan threshold"? One threshold is when you simply can't figure out what the function is doing. But most programs decay gradually; they don't become instantly opaque. How can you observe that decay and measure it?

The reason you don't see the decay is that you probably remove all traces of maintenance. If you are like most programmers, when you delete lines, you actually delete them. When you change lines, you probably don't leave the originals behind. When you add lines, if you work like most people, you leave little indication that they weren't in the original. Here is a simple convention that will give a complete audit trail of maintenance.

1) When you delete a line, don't actually delete it. Simply insert a lamp symbol (⍝) in front of it, and turn it into a comment.

2) When you change a line, don't change the original. First turn the original line into a comment by inserting a lamp symbol as in step 1. Then make a copy of the original right below itself. (Don't type it in again. Use the function editor to do the copying. Even the most primitive ∇ editor will allow you to make copies if you edit the line number within the brackets.) Modify the copy, and when you are happy with it, remove the lamp on the copied line.

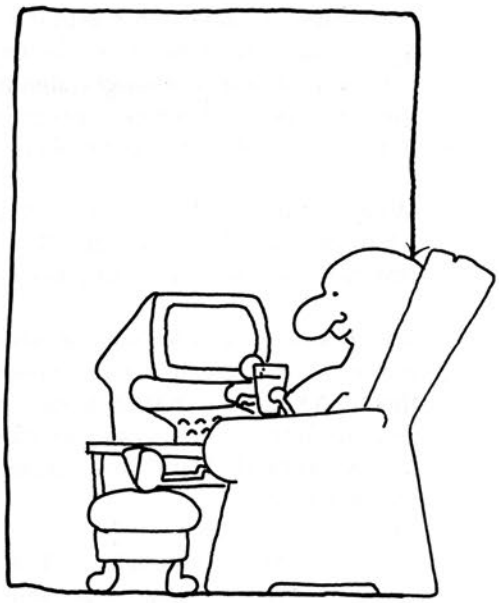
3) When you insert a line, always include a comment which marks it as an addition. The easiest thing to do is to include a comment at the end of the line which just has the plus (+) symbol. If you are not working on an implementation like SHARP APL which allows line-end comments, put the comment tag immediately above the added line.

Now you will have a complete record of all the maintenance done on the program. When the percentage of audit trail comments exceeds some threshold (perhaps 25% or 33%), you should start a new version from scratch.

You will find that this technique will do precisely what it is intended to do—make the decay of a program painfully obvious. If you can't stand the sight of a rotting program, you may want to use a more sanitary method. Instead of leaving comments which contain the old code, you can leave comments which just indicate who made the change and when. The most common practice is to include the person's initials, the date of the change, and perhaps the type of change (Add, Change, Delete). You will still be able to determine the percentage of the statements which are changes. The abbreviated form, however, will probably be a lot less distracting.

The purpose of this section has been to introduce a practical approach to treating APL functions as interchangeable parts. If the principle is appropriate for hardware engineering, we should consider its value for software engineering as well.

Making Life Easier



There are simple things you can do to increase the quality of your programs with respect to ease of modification. Essentially, you need to avoid some common practices which will cause you extra work when you change your programs.

Branching targets should always be selected from one or more labels. You should always avoid branching to absolute line numbers. Some of the more common patterns of branching to absolute line numbers are listed below.

→linenumber
→linenumber×⍳condition
→(condition)⍴linenumber
→(condition)↑linenumber
→(condition)/linenumber

The reason that you should not branch to absolute line numbers is that the correct target number will change when you add or delete lines. Unfortunately, your branch statements won't change. If you do make such changes, you will have twice the work to do. First, you add or delete the lines. Then you have to go back and change all the branch statements which are affected. Why do twice the work? In addition, you should know that on implementations like SHARP APL, branching to line numbers instead of labels is somewhat slower than branching to labels.

You should also avoid relative branch statements. By this we mean computing a branch target as an offset from the Line Counter or a label. Some of the more common patterns of relative branching are listed below.

→constant +or- *⎕LC*
→*⎕LC* +or- **constant**
→(condition)⍴*⎕LC* +or- **constant**
→(condition)↑*⎕LC* +or- **constant**
→(condition)/*⎕LC* +or- **constant**
→*⎕LC*
→*⎕LC*×⍳**condition**
→(condition)⍴*⎕LC*
→(condition)↑*⎕LC*
→(condition)/*⎕LC*
→constant +or- **label**
→label +or- **constant**
→(condition)⍴label +or- **constant**

→(condition)↑label +or- constant
→(condition)/label +or- constant

The problem with relative branching is the same as with branching to absolute line numbers. The branch targets will become invalid if you insert or delete lines between the branch line and the target line.

I find people who normally use labels succumb to the temptation to use relative branching in one particular situation. This happens when they are working on an implementation like SHARP APL, which provides the diamond (◇) statement connector. The temptation is to write IF-THEN constructs like the following:

→(condition)ρ⎕*LC*+1 ◇ **statement ◇ statement** ...

Some people might find this harmless, but it has exactly the same problem as any other kind of relative branching. If enough statements are added to the THEN part, a line will have to be inserted between the branch statement and the branch target line. Then the branch statement itself will have to be changed. If you want to minimize the work you do in modifying programs, branch to line labels only.

Conditional execute presents problems similar to branching. The typical conditional execute statement looks like the following:

⍎(condition)/'statement'

If the diamond statement connector is available, you may also see the following sort of expression:

⍎(condition)/'statement ◇ statement ◇ ...'

In the first case, if the THEN part of this IF-THEN construct subsequently requires more than one statement, the entire construct will have to be completely redone to use a branch. In either case, if an ELSE part must be added later, the entire construct will have to be broken up into several statements using branches. Why not anticipate change and avoid the conditional execute altogether? In addition, you should be aware that using conditional execute is usually somewhat slower than using branches to do the same thing.

The next practice to avoid is really a whole set of related practices. Avoid defining functions in such a way that they are difficult to edit. The specifics of this admonition vary with the implementation you are using. If the editor you have available makes it awkward to edit lines longer than the terminal width, don't write lines longer than that width. If the editor you have available makes it awkward to edit lines which contain literal constants which include embedded carriage returns, don't create such constants. If anything will be difficult to change, don't use it!

The final practice you should avoid is the re-use of variable names. By this we mean having more logical variables than variable names. This sort of coding is only possible with a language like APL, in which you can assign arrays of different rank, shape, and data type to the same name in the same program. We see some APL programmers assigning completely unrelated values to the same name, particularly in long programs.

There are probably a couple of reasons why people do this. First, if the program is quite long, the list of local variables will probably spill over a single line. In order to keep the header on one line, they re-use names. Others may be facing workspace size limitations. Once again, in a large program, there may be a lot of local variables lying around that will not be used again. In order to conserve space, these variables are re-assigned values that are needed.

Re-using names is not the solution to these problems. The solution is to write shorter programs. If the long function is broken into several subfunctions, the long header will also be split among several headers. If the long function is broken up, APL will automatically make available the space which was occupied by local variables which are no longer in use.

Some programmers like to use a single variable as a "sinkhole" or "trashcan" variable. Such a variable is assigned results that aren't needed, usually to prevent them from being printed. This will result in many unrelated values being assigned to this special variable. Some examples of such a variable are listed below:

```
[⍳] 'ALIGN PAPER'
[⍳] SINK←⍞

[⍳] JUNK←⎕FX MATRIX
```

Such a practice seems relatively harmless, as long as there is only one such variable, and the values it is assigned are truly not needed.

The re-use of variable names can be a source of great frustration when modifying programs. If you are revising a program, and decide that you need a value which has been overwritten, you are going to have to go back and change all references to that variable which occur after the re-assignment. Why make trouble for yourself? You can see that reusing names in this way makes debugging much more difficult, and nullifies much of the usefulness of the program stop facility.

The person who can do the most to make life easier for you is yourself. In the world of programming, premeditated laziness is a virtue. Don't make unnecessary work for yourself, or the people you work with.

VALIDATION

Once you have found what you need to change, and have made that change, you need to validate that it works. This isn't always done. When the first draft of this paper came back from typing, I found that the definition of validation had been typed as "assume that the program works". While I believe that I actually wrote "assure that the program works", I am afraid that the typo more accurately reflects what many programmers do.

Entering Test Input

In designing a maintainable system, there are three steps you can take to reduce the quantity of data that must be entered or validated. The first of these is to include in your file design a set of test case files which are always kept online. You should never test a change on production files. Rather than make copies from production files every time you want to test, keep a set of small test case files always available.

This concept presumes that you design the system so that it is relatively easy to switch the files being used. A common convention is to refer to a file by a global variable with the same name as the file. This variable would contain the file tie number if you were using SHARP APL. This enables you to write expressions like the following:

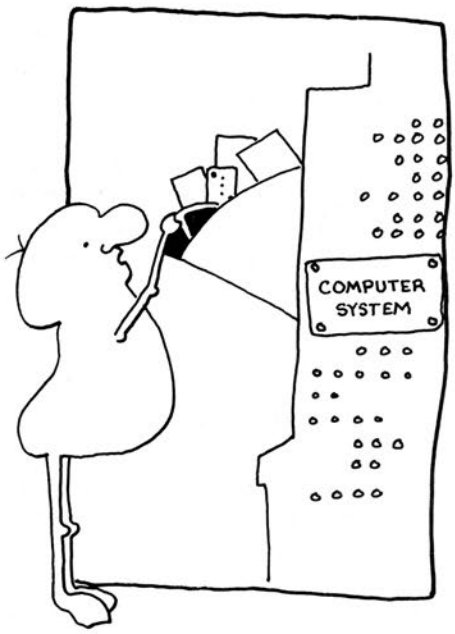
```
⎕READ MASTER,COMPONENT
DATA ⎕REPLACE TRANSACTIONS,COMPONENT
```

You can also write a trivial function to switch to test files:

```
    ∇ SETUPTEST
[1]  MASTER←OPENFILE 'MASTERTEST'
[2]  TRANSACTIONS←OPENFILE 'TRANSTEST'
    ∇
```

In some cases, it may be preferable to generate test data as needed rather than preserve test cases. If this is true, you can include the second feature to reduce the amount of test data that must be entered manually. This is a test data generator. It can work in two ways. It can generate test files completely from scratch. Perhaps, more useful, it can select a randomly chosen subset of information from the actual files.

The third feature you can build into a system to reduce the quantity of data that needs to be entered during a test is a script file. If you have been using a single input utility function to get input from a terminal, this will be easy to implement. You merely need to modify that function to check the value of a global variable, say *TESTSCRIPT*. If it is 0, things go on as usual. But if it is non-zero, it is assumed to be the component number of the test script file which should be read when the next terminal input is requested. When the file is read, the variable is incremented for the next usage. If you are using SHARP APL, you might make this variable contain both the file tie number and component number. This will allow you to use more than one script file. The segment of your input utility might look like the following:




```
[∘]  →(TESTSCRIPT∧.=0)ρKEYBOARD
[∘]  INPUT←⎕READ TESTSCRIPT
[∘]  TESTSCRIPT←TESTSCRIPT+0 1
[∘]  →VALIDATE
[∘]  KEYBOARD:⍞←PROMPT,' '
[∘]  INPUT←(1+ρPROMPT)↓⍞
```

If you are using SHARP APL, you can also use an S-task to accomplish the same purpose.

If you use both test case files and a script file, testing really becomes easy. You switch data files, open the script file, set the script counter, execute *⎕LX*, and go to lunch!

Checking Test Output

You can also design features into your system which will reduce the amount of data that has to be checked once the test has been run. There are five features which can help you with this, which you may want to design into your application.

When we speak of validating the output of a test, you probably think of checking the results of a report. But many programs do not produce any printed output. You can design your application with the testing of these programs in mind.

If you are using a complicated file structure you should probably write a special program which will validate the structure of that file. It should check for the following sorts of problems and print the component numbers where there are any problems:

1) Directories are the right shape
2) All components pointed to actually exist
3) Pointers point to the right component

4) Back pointers get you back to where you started

The program suggested above validates the *structure* of the file. Even when you have a simple file structure, you can benefit from a program which checks the *values* in a file. It should check for the following types of problems and print the numbers of components where they occur:

1) Arrays whose internal representation has changed, such as fixed point arrays which have become floating point

2) Arrays or fields whose values are not members of specified sets, such as invalid dates

These two kinds of programs can save you days in finding obscure bugs. They are much more helpful than simply printing out the file. Obviously, they are used for one aspect of testing that many people ignore. Not only should you test that a program did what it *was* supposed to, but also test that it didn't do anything it *was not* supposed to. These functions help with checking the latter problem.

You can take a slightly different approach to the same problem. Instead of validating the file after the fact, you can do so as it is being processed. If you are using cover functions for your files, you can do the content validation as the information is stored on or retrieved from the file. You will want to provide some sort of switch which can turn the validation on and off. For reasons of efficiency, you may not want to have the validation code operating during production runs.

You can extend this concept one step further and have exhaustive validation of every argument of each function. This requires extra coding for the application programs, but it is frequently well worth it. Once again, you may want to be able to turn the validation on and off. This can be accomplished by a simple test and branch at the beginning of the program.

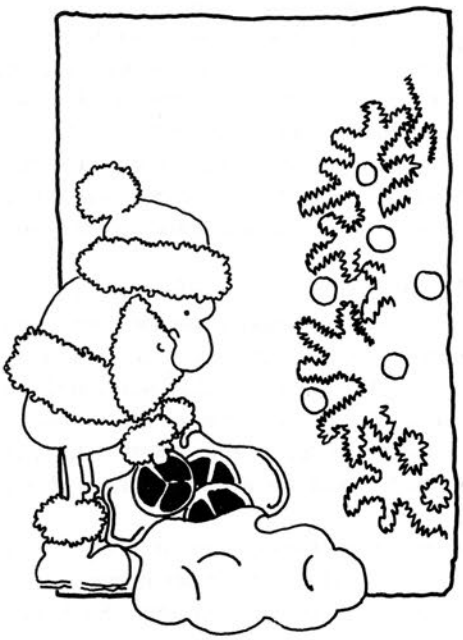
A more sophisticated method involves using executable comments. In this case, the code which validates the arguments is written on comment lines. These are specially marked to indicate that they can be converted into executable lines by a special program. A simple convention that can be followed is that if the first character of a comment is a hydrant (⍏) symbol, the line can be converted into executable code by simply rotating the first two symbols on the line to the end of the line. A function which will convert all such lines into executable statements can be found in the Appendix.

While these approaches are oriented towards finding behaviours that the application ought not to exhibit, you also need help in making

sure that the programs did what they were supposed to. If you are checking file updates, probably the biggest help is a simple file comparator. It takes two files and prints information about those components where the files differ. If your program has worked properly, when you compare the test file with the original the only components which should differ should be the ones which were updated. You can concentrate on looking at the components flagged by the comparator and ignore the rest. If your application changes global variables, you may also want to use a function which compares the variables in two workspaces.

These suggestions provide a number of ways you can design features into your applications so that maintenance is made easy. Validation is very important, but there is every reason to look for ways to minimize the amount of work needed to get the degree of certainty we want.

DISTRIBUTION



Having found what needs to be changed, made the change, and tested to ensure that the changes work, we are now ready to distribute the changed software. How much effort this will actually require varies greatly depending on the structure of the application and who is using it. We can distinguish two variables which control this effort. First, applications are typically structured in three ways: single workspace, multiple workspaces, functions stored on file. In addition, the number of distinct installations (machines) that the application is running on determines what you will need to do to distribute the software.

Distribution involves three main steps:

1) Preserve a copy of the old version.
2) Install the new version.
3) Notify the users of the change.

How you preserve a copy of the change depends on the structure of the application. If this is a single workspace application, you can just load the workspace, rename it, and save it. If the function you have changed is used in several workspaces, you will have to repeat this procedure for each workspace. If the functions are stored on file, you might follow two courses. If the changes affected a relatively small number of functions, you can do the following:

1) Fetch the old functions from file.
2) Make copies of them with new names.
3) Store the old copies on the file.

If many changes have been made, you may prefer to make a copy of the entire file and give it a new name.

The reason for making a backup copy should be obvious. Even though you have tested your change, it may still have problems. You need to go back to the old version if production usage of the application is to continue while you fix your change. If you are using a product like SHARP APL which allows incremental backup of files and workspaces, the system managers will probably do a daily backup of all changes made to all files and workspaces. If you need a copy of one of your workspaces or files from a previous day, you can ask the operator, and soon your old system will be back on the computer. This will suffice if you only make one change per day. But if you make two separate changes in a single day, the retrieved copy will only show the last version. You should carry a copy of every changed version that was made on any one day. Then you will always be able to go back to an old version if necessary.

Installation follows a procedure similar to that of preservation. You must access the experimental version, rename it to become the production version, and store it in the

appropriate place. This will suffice if you are working on a single machine. But if you are distributing software to many sites, you will need to copy the new version onto a physical medium (probably magnetic tape). Then you will transport it to the other sites, and have it installed. If the target system is not running the same APL implementation and version as the source system, you have an additional set of problems to face. You will have to use a source level transfer mechanism like the Workspace Interchange Standard defined by the ACM Special Interest Group on APL (SIGAPL). The standard makes the transfer possible. It does not make it easy.

Finally, you must notify the users of the change you have made. It is definitely poor etiquette to subject unsuspecting users to changed software. One of the most commonly used schemes for notifying users of "system news" is based on $\square LX$. Among the functions called by $\square LX$, one checks a special file which contains "news messages". If the user who has loaded the workspace has not seen the news messages, he is notified of their existence and asked if he wishes to see them. Of course, if you only have one person using an application, a phone call will suffice.

There are several special problems associated with distribution. For reasons of space efficiency, it is sometimes necessary to compile a "production version" from the maintenance copy of your system. Such conversions typically include:

1) Replacing variable names with a small set of short names
2) Placing multiple statements on a single line
3) Deleting comments

Because such production versions are harder to understand, it is important to do maintenance on the maintenance copy only. In fact, it may be wise to actually lock the production version to prevent patching. This brings up a related problem. If for security reasons you lock your functions before distributing them, you must store an unlocked copy before doing the locking. Otherwise, you will have solved your maintenance problems by making your only copies unmaintainable!

CONCLUSION

APL can be a maintainable programming language. You have seen techniques for making APL systems easier to maintain in each of the four phases of maintenance: diagnosis, modification, validation, and distribution. These are proven techniques you can put into practice.

If you use APL professionally, your programming time is a precious commodity. As the saying goes, "time is money". If you use the ideas presented in this paper, you will save both your own time and the money of whoever ultimately bears the cost of your programming efforts. APL can make you a very productive developer of new programs. If you design your systems right, you can also be a productive maintainer of existing programs.

REFERENCES

1. DeKerf, J.L.F. "APL and Halstead's Theory of Software Metrics." *APL81 Conference Proceedings.* New York: Association for Computing Machinery, 1981.

2. Falkoff, A.D., and K.E. Iverson. "The Design of APL." *IBM Journal of Research and Development*, Vol. 17, No. 4, (July 1973).

3. Halstead, M.H. *Elements of Software Science.* New York: Elsevier-North Holland, 1977.

4. Iverson, K.E. *A Programming Language.* New York: John Wiley and Sons, 1962.

5. Kernighan, B.W. and P.J. Plauger. *Software Tools.* Reading, MA: Addison Wesley Publishing Co., 1976.

6. Mauri, R.A., and A.H. Williams. "Extending Halstead's Software Science for a More Precise Measure of APL." *APL82 Conference Proceedings.* New York: Association for Computing Machinery, 1982.

7. McGrew, J. "Programmer Productivity Gains through Emphasis of Fundementals." *Proceedings, 1980 APL Users Meeting.* Toronto: I.P. Sharp Associates Limited, 1980.

8. McLean, E. "The Use of APL for Production Applications." *APL76 Conference Proceedings.* New York: Association for Computing Machinery, 1976.

9. Parnas, D.L. "On the Criteria to be Used in Decomposing Systems into Modules." *Communications of the ACM,* Vol. 15, No. 12, (December 1972).

10. Shneiderman, B. *Software Psychology.* Cambridge, MA: Winthrop Publishers, 1980.

11. Weinberg, G.M. *The Psychology of Computer Programming.* New York: Van Nostrand Reinhold, 1971.

APPENDIX

```
   ∇ TIE←KSTART NAME
[1] ⍝ STARTS A NEW FILE OF KEYED ACCESS RECORDS
[2] ⍝ ARGUMENT- CHAR VECTOR NAME OF FILE
[3] ⍝ EXAMPLE- DB←KSTART 'DATABASE'
[4]  TIE←1+⌈/0,⎕NUMS
[5]  NAME ⎕CREATE TIE
[6]  ⍞←'INDEX',(⍕TIE),'←0 2⍴0'
[7]  (⍞←'INDEX',⍕TIE) ⎕APPEND TIE ∇

   ∇ TIE←KOPEN NAME
[1] ⍝ OPENS AN EXISTING FILE OF KEYED ACCESS RECORDS
[2] ⍝ ARGUMENT- CHAR VECTOR NAME OF FILE
[3] ⍝ RESULT- INTEGER SCALAR FILE TIE NUMBER
[4] ⍝ EXAMPLE- DB←KOPEN 'DATABASE'
[5]  TIE←1+⌈/0,⎕NUMS
[6]  NAME ⎕STIE TIE
[7]  ⍞←'INDEX',(⍕TIE),'←⎕READ TIE,1' ∇

   ∇ KCLOSE TIE
     ;JUNK
[1] ⍝ CLOSES A FILE OF KEYED ACCESS RECORDS
[2] ⍝ ARGUMENT- FILE TIE NUMBER
[3] ⍝ EXAMPLE- KCLOSE DB
[4]  JUNK←⎕EX 'INDEX',⍕TIE
[5]  ⎕UNTIE TIE ∇

   ∇ R←T KEY K
     ;⎕IO
[1] ⍝ CONVERTS A LITERAL KEY TO ITS NUMERIC EQUIVALENT
[2] ⍝ RIGHT_ARGUMENT- CHAR VECTOR KEY (NOT MORE THAN 9 ELEMENTS)
[3] ⍝ LEFT_ARGUMENT- TIE NUMBER OF FILE
[4] ⍝ RESULT- LEFT ARGUMENT CATENATED WITH CONVERTED RIGHT ARGUMENT
[5] ⍝ NOTE- THIS FUNCTION USED ONLY IN CONJUNCTION WITH KPUT, KGET, KDELETE
[6]  ⎕IO←0
[7]  R←T,37⊥' ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'⍳(9⌊⍴K)↑K ∇

   ∇ PTR←INDEX KFIND ID
     ;⎕CT
[1] ⍝ FINDS THE COMPONENT WHICH CONTAINS RECORD(S) RELATED TO A SPECIFIED KEY
[2] ⍝ RIGHT_ARGUMENT- INTEGER SCALAR KEY
[3] ⍝ LEFT_ARGUMENT- N×2 NUMERIC DIRECTORY OF KEYS AND COMPONENTS
[4] ⍝ RESULT- INTEGER SCALAR SPECIFYING COMPONENT NUMBER
[5] ⍝ NOTE- THIS FUNCTION USED ONLY AS A SUBFUNCTION OF KPUT, KGET, KDELETE
[6]  ⎕CT←0
[7]  PTR←(INDEX[;2],0)[INDEX[;1]⍳ID] ∇
```

```
   ∇ DATA KPUT KEY
     ;TIE;PTR
[1] ⍝ ADDS OR REPLACES DATA IN A KEYED ACCESS FILE
[2] ⍝ RIGHT ARGUMENT- FILE TIE NUMBER AND KEY
[3] ⍝ LEFT ARGUMENT- DATA ARRAY TO BE STORED
[4] ⍝ EXAMPLE: RECORD KPUT DB KEY 'ROCH'
[5]  TIE←1↑KEY
[6]  PTR←(⍎'INDEX',⍕TIE) KFIND 1↓KEY
[7]  →(0=PTR)/APPEND
[8] REPLACE:DATA ⎕REPLACE TIE,PTR
[9]  →EXIT
[10]APPEND:PTR←DATA ⎕APPENDR TIE
[11] ⍎'INDEX',(⍕TIE),'←INDEX',(⍕TIE),',[1] (1↓KEY),PTR'
[12] (⍎'INDEX',⍕TIE) ⎕REPLACE TIE,1
[13]EXIT: ∇

   ∇ DATA←KGET KEY
     ;PTR
[1] ⍝ RETRIEVES DATA STORED IN A KEYED ACCESS FILE
[2] ⍝ ARGUMENT- FILE TIE NUMBER AND KEY
[3] ⍝ RESULT- DATA ARRAY STORED UNDER THAT KEY
[4] ⍝ EXAMPLE: RECORD←KGET DB KEY 'BOST'
[5]  PTR←(⍎'INDEX',⍕1↑KEY) KFIND 1↓KEY
[6]  →(0≠PTR)/READ
[7]  ⎕←'INVALID KEY'
[8]  →EXIT
[9] READ:DATA←⎕READ(1↑KEY),PTR
[10]EXIT: ∇

   ∇ KDELETE KEY
     ;PTR;NAME
[1] ⍝ DELETES RECORD FROM A KEYED ACCESS FILE
[2] ⍝ ARGUMENT- FILE TIE NUMBER AND KEY
[3] ⍝ EXAMPLE- KDELETE DB KEY 'CHIC'
[4]  NAME←'INDEX',⍕1↑KEY
[5]  PTR←(⍎NAME) KFIND 1↓KEY
[6]  ⍎NAME,'←((0 ¯1↓',NAME,')∨.≠1↓KEY)⌿',NAME
[7]  (⍎NAME) ⎕REPLACE(1↑KEY),1
[8]  '' ⎕REPLACE(1↑KEY),PTR ∇
```

```
    ∇ TABLE←PERMISSION EDIT△TABLE TABLE
      ;RANK;COMMANDS;ACTION;TEMP;ENTRY;C
ONTROL;ARGUMENT;EXPAND;SELECT;INDEX;EXPRESSION;ALPHABET;AXIS;INPUT;CONVERT
[1] ⍝ ALLOWS USER TO EDIT DATA TABLE
[2] ⍝ RIGHT ARGUMENT- VECTOR OR MATRIX
TABLE TO BE EDITED. VECTOR TREATED AS 1-ROW MATRIX
[3] ⍝ LEFT ARGUMENT- PERMISSION CODES F
OR USER ACTIONS. 1 ROW,2 COLUMN, 4 ELEMENT ORIENTED COMMANDS
[4] ⍝ RESULT- TABLE TABLE ALTERED AS SPECIFIED BY THE USER
[5] ⍝ SUBROUTINES- EDIT△LINE
[6] ⍝ EXAMPLE: 1 EDIT△TABLE 3 4⍴'WOLFBEARLION'
[7]  PERMISSION←1=⍬(4⍴2)⊤⌊''⍴PERMISSION
[8]  RANK←⍴⍴TABLE
[9]  ALPHABET←⎕AV[86+⍳54]
[10] TABLE←(1⌈¯2↑⍴TABLE)⍴TABLE
[11] CONVERT←(0=1↑0⍴TABLE)/'ARGUMENT←⎕FI ARGUMENT'
[12] COMMANDS← 11 2 ⍴'ARDRIRERPRACDCICEDPCCE'
[13]LOOP:⎕←TEMP←'ACTION? '
[14] ENTRY←' ',(⍴TEMP)↓⍞
[15] →('?*'=1↑1↓ENTRY)/HELP,EXIT
[16] ARGUMENT←(ENTRY⍳':')↓ENTRY
[17] ENTRY←(-(0≠⍴ARGUMENT)+⍴ARGUMENT)↓ENTRY
[18] ⍎CONVERT
[19] CONTROL←(⎕VI ENTRY)/⎕FI ENTRY
[20] ACTION←((ENTRY∊ALPHABET)∧¯1↓0,ENTRY∊' ')/ENTRY
[21] ACTION←(COMMANDS∧.=2↑ACTION)⍳1
[22] →(ACTION≤1↑⍴COMMANDS)⍴OK
[23]HELP:'*** ACTIONS ARE ***'
[24] ⎕←'   ADD ROW- A R:<VALUE>                ADD COLUMN- A C:<VALUE>'
[25] ⎕←'DELETE ROW- D R <NUMBER>             DELETE COLUMN- D C <NUMBER>'
[26] ⎕←'INSERT ROW- I R <NUMBER>:<VALUE>     INSERT COLUMN- I C <NUMBER>:<VALUE>'
[27] ⎕←'  EDIT ROW- E R <NUMBER><POSITION>     EDIT COLUMN- E C <NUMBER> <POSITION>'
[28] ⎕←' PRINT ROW- P R <NUMBER>              PRINT COLUMN- P C <NUMBER>'
[29] ⎕←'CHANGE ELEMENT- C E <ROW> <COLUMN>: <VALUE>'
[30] →LOOP
[31]OK:AXIS←(⌈ACTION÷5)-~⎕IO
[32] →(AXIS=3)⍴CHECK
[33] EXPRESSION←'TABLE[',((⎕IO-AXIS)⍬'INDEX;'),']'
[34]CHECK:→(PERMISSION[AXIS])⍴GO
[35] ⎕←'*** UNAUTHORIZED COMMAND ***'
[36] →EXIT
[37]GO:→((10⍴ADD,DELETE,INSERT,EDIT,PRINT),CHANGE)[ACTION]
[38]ADD:INPUT←(⍬⍴TABLE)[AXIS]↑ARGUMENT
[39] TABLE←TABLE,[AXIS] INPUT
[40] →LOOP
[41]DELETE:INDEX←⎕IO⌈((⍴TABLE)[AXIS]-~⎕IO)⌊1↑CONTROL
[42] SELECT←(⍴TABLE)[AXIS]⍴1=1
[43] SELECT[INDEX]←0
[44] TABLE←SELECT/[AXIS] TABLE
[45] →LOOP
[46]INSERT:INDEX←1+(⎕IO-1)⌈(⍴TABLE)[AXIS]⌊1↑CONTROL
```

```
[47] EXPAND←(1+(ρTABLE)[AXIS])ρ1=1
[48] EXPAND[INDEX]←0
[49] TABLE←EXPAND\[AXIS] TABLE
[50] INPUT←(⌽ρTABLE)↑ARGUMENT
[51] ⍎EXPRESSION,'←INPUT'
[52] →LOOP
[53]EDIT:INDEX←⎕IO⌈((ρTABLE)[AXIS]-~⎕IO)⌊1↑CONTROL
[54] INPUT←(1↑1↓CONTROL) EDIT∆LINE⍕,⍎EXPRESSION
[55] ⍎CONVERT
[56] INPUT←(⌽ρTABLE)[AXIS]↑INPUT
[57] ⍎EXPRESSION,'←INPUT'
[58] →LOOP
[59]PRINT:INDEX←⎕IO⌈((ρTABLE)[AXIS]-~⎕IO)⌊1↑CONTROL
[60] ⎕←⍕⍎EXPRESSION
[61] →LOOP
[62]CHANGE:INDEX←⎕IO⌈((ρTABLE)-~⎕IO)⌊CONTROL
[63] TABLE[1↑INDEX;1↓INDEX]←1↑ARGUMENT
[64] →LOOP
[65]EXIT:⍎(1=RANK)/'TABLE←,TABLE' ∇

   ∇ TEXT←POSITION EDIT∆LINE LINE
     ;INPUT;TEMP;QUIT;LENGTH;LOCATION;INSERT;⎕IO
[1] ⍝ ALLOWS USER TO EDIT A LINE OF TEXT IN A MANNER SIMILAR TO ∇ EDITING
[2] ⍝ COMMANDS: '/' DELETE; '\' DELETE
AND INSERT BLANK; '0'-'9' INSERT BLANKS AS SPECIFIED
[3] ⍝            'A'-'Z' INSERT BLANKS-
5 TIMES THE POSITION OF THE LETTER IN THE ALPHABET
[4] ⍝            ',' INSERT ALL TEXT TO
THE RIGHT OF COMMA HERE; '.' INSERT ALL TEXT TO THE RIGHT OF PERIOD HERE
AND QUIT EDIT
[5] ⍝ RIGHT ARGUMENT- CHAR VECTOR, LINE OF TEXT TO BE EDITED
[6] ⍝ LEFT ARGUMENT- INTEGER SCALAR SPE
CIFYING POSITION AT WHICH PRINTING ELEMENT SHOULD BE PLACED
[7] ⍝ RESULT- ALTERED CHARACTER VECTOR
[8]  QUIT←0
[9]  ⎕IO←1
[10] →(0≠POSITION)ρDISPLAY
[11] ⍞←LINE
[12] TEXT←⍞
[13] →EXIT
[14]DISPLAY:⎕←TEXT←LINE
[15] ⍞←(POSITION-1)ρ' '
[16] INPUT←⍞
[17] →(~∨/',.'∈INPUT)ρDELETE
[18] LOCATION←¯1+⌊/INPUT⍳',.'
[19] TEXT←(LOCATIONρTEXT),((LOCATION+1)↓INPUT),LOCATION↓TEXT
[20] QUIT←'.'=INPUT[LOCATION+1]
[21] INPUT←LOCATIONρINPUT
[22]DELETE:LENGTH←(ρTEXT)⌈ρINPUT
[23] TEXT←('/'≠LENGTH↑INPUT)/LENGTH↑TEXT
[24] INPUT←('/'≠LENGTH↑INPUT)/LENGTH↑INPUT
[25] TEXT[(INPUT='\')/⍳ρTEXT]←' '
```

```
[26] INSERT←(0 0 ,(⍳9),(5×⍳26),0)[(' 0123456789',⎕AV[86+⍳26])⍳INPUT]
[27] TEXT←((⍳(⍴INSERT)++/INSERT)∈(⍳⍴INSERT)++\INSERT)\TEXT
[28] →QUIT⍴EXIT
[29] POSITION←1↑(' '≠INPUT)/⍳⍴INPUT
[30]ALTER:POSITION←(¯1+(1+⍴TEXT)×0=POSITION)+POSITION
[31] ⍞←TEMP←TEXT,((⍴TEXT)-POSITION)⍴⎕AV[159]
[32] INPUT←(POSITION⍴' '),(⍴TEMP)↓⍞
[33] LENGTH←(⍴INPUT)⌈⍴TEXT
[34] INPUT←LENGTH↑INPUT
[35] TEXT←LENGTH↑TEXT
[36] LOCATION←((INPUT≠' ')∧TEXT=' ')/⍳⍴INPUT
[37] TEXT[LOCATION]←INPUT[LOCATION]
[38]EXIT: ∇

   ∇ SWITCH ROTATECOMMENTS NAMES
     ;LABEL;CTR;LMT;FNREP;JUNK
[1] ⍝ ROTATES ⍝⍎ COMMENTS SO THAT THE COMMENT TEXT BECOMES A STATEMENT OR VICE VERSA
[2] ⍝ RIGHT ARGUMENT- MATRIX NAMELIST OF FUNCTIONS TO BE CHANGED
[3] ⍝ LEFT ARGUMENT- 1 (COMMENTS TO STATEMENTS), OR 0 (STATEMENTS TO COMMENTS)
[4]  LABEL←(SWITCH,~SWITCH)/RIGHT,LEFT
[5]  CTR←0
[6]  LMT←1↑⍴NAMES
[7] LOOP:CTR←CTR+1
[8]  →(LMT<CTR)⍴END
[9]  FNREP←⎕CR NAMES[CTR;]
[10] →LABEL
[11]RIGHT:JUNK←⎕FX(2×(((1↑⍴FNREP),2)↑FNREP)∧.='⍝⍎')⌽FNREP
[12] →LOOP
[13]LEFT:FNREP←(1-(FNREP=' ')⊥1)⌽FNREP
[14] JUNK←⎕FX(¯2×(((1↑⍴FNREP),¯2)↑FNREP)∧.='⍝⍎')⌽FNREP
[15] →LOOP
[16]END: ∇
```

DESIGNING EFFICIENT APL SYSTEMS

Joshua S. Levine
Product Manager
Applications Software
I.P. Sharp Associates Limited
Toronto, Ontario

For four wicked centuries the world has dreamed this foolish dream of efficiency; and the end is not yet.
- Bernard Shaw, John Bull's Other Island, Act IV

We're all interested in saving time, money, and effort. If not, we should be. People use computers to make more efficient use of their resources, thus increasing their productivity.

Efficiency is the capacity to produce results with a minimum expenditure of energy, time, money, or materials. Several factors determine an operation's efficiency:

- computer resources consumed
- physical data storage required
- design and implementation time
- input and output volume

Thus, an application is most efficient when it minimizes these factors; though all are related and optimizing one might affect another.

Other requirements also influence an application's efficiency. For example:

- application security
- "SYSTEM CRASH" recovery
- maintainability
- user-friendliness

These are not the focus of this discussion, though they are significant when considering an APL system design.

The traditional approach to the topic of efficiency and APL has been a discussion of techniques to optimize an application *after the fact*; after the system has been implemented. Experience gained in the initial development helps to point out areas for improvement and optimization, but the ability to optimize an existing system at a later time depends on the initial design. If a system runs slowly and expensively, the user may decide to abandon it. Therefore, time spent at the outset makes future optimization possible.

The most important basis for an efficient design is to know what the system is supposed to do and how it will be used. These are simple concepts, yet many applications are written with just a cursory idea of how it will work and what it will be used for. Compiled programming languages force programmers to analyze all aspects of the application as a matter of course. APL is more flexible in this regard. APL makes it possible to specify a system by creating a prototype, which is a working specification. Specifications, either written or in APL, are needed to ensure that the user knows what he wants and what he will receive. Good specifications contain:

- a general overview of the system
- a description of the method of data input
- a description of the actions to be performed
- a description of the method of user interaction
- a description of the method of data output

Once these have been outlined, general efficiency decisions can be addressed. Among other things, limitations which affect overall efficiency can be noted and explained to the user.

With a working knowledge of the intended uses, major decisions can be easily made regarding:

- data and program structure
- data and program storage requirements
- input/output mechanisms

The following sections discuss the topic of efficiency. They present various methods for identifying problem areas and for improving program productivity.

STRUCTURES explains specific techniques for structuring and organizing data and functions in an efficient manner.

ALGORITHMS discusses efficiency at the APL primitive level.

MEASUREMENT provides a set of tools which can be use to measure the relative efficiency of primitives and algorithms.

I do not consider that efficiency need be mated
to extreme delicacy or precision of touch...
It should possess a sweeping gesture - even if that gesture
may at moments sweep the ornaments from the mantelpiece.
- Harold Nicolson, Small talk: On Being Efficient

STRUCTURES

Data Structures and Files

Often we must design an application which stores data outside a workspace. For those not in a virtual storage environment, the reasons are obvious. Even when very large workspaces are available, asking the user to *)SAVE* often, or at the end of a session, is an unreliable means of storing data permanently. In addition to space constraints and reliability concerns, we must consider needs for data sharing, data integrity, and data security. What if different users need to be able to use the data, or update parts of it simultaneously, or should have access to only part of it? These are common requirements in application systems. Therefore, most APL implementations provide some way to store data external to a workspace. Such external data collections are called "files".

Several APL systems, such as SHARP APL, have file subsystems which are accessed by primitive system functions. Other systems use shared variables to access externally stored data. In both cases, these systems provide files which are organized into components that can be accessed in random order. These components contain single APL data objects which can be used like any other APL data. The discussion which follows presumes the use of such a component file system.

There are many ways of structuring data for a given application, and each one has specific advantages and disadvantages. This section of the paper explores two complementary ways of storing data: inverted and uninverted.

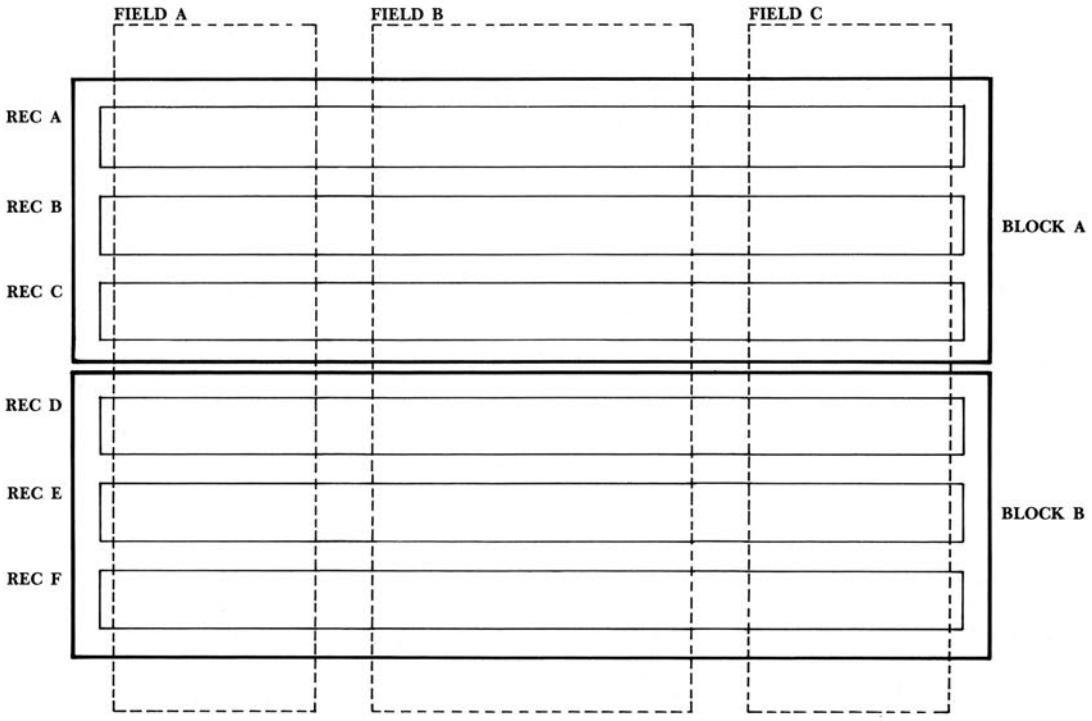
Uninverted Design

"Uninverted" may seem to be a rather odd word. We are using an uninverted structure when we store two-dimensional data (records by fields) in record-major order.

To analyze the performance of this structure, we will construct a sample data base of *REC* records and *FLD* fields. In order to simplify the problem, we will assume that each element of the data, *DATABASE*[*REC*;*FLD*], is a typeless scalar, regardless of actual internal type or shape. In practice, of course, if the elements have different data types, or different shapes, we either split each block into several components or use features like enclosed arrays to combine unlike objects.

If the entire data base does not fit in a workspace, we need to partition it into smaller arrays. These blocks of records are stored in components of the file. Conceptually, this partitioning is applied along the first dimension of the data, such that each block contains *REC∆BLK* records and *FLD∆BLK* fields. Figure 1 displays the physical layout of the uninverted design.

Figure 1



Record Retrieval - Uninverted Design

How efficient is an uninverted structure? How many records should we store in each block for maximum efficiency? To answer these questions, we must analyze the relative efficiency of the various possibilities as they will be used.

We will focus on three aspects of retrieval. These are the costs associated with reading blocks, executing APL loops and extracting records [Note 1]. We can calculate these cost segments in terms of the following constants:

COST∆BUF - cost of reading a buffer from file
SIZE∆BUF - size of a file system record; currently 3156 bytes on the SHARP APL system [Note 2]
COST∆ROW - cost of indexing elements rowwise
COST∆COL - cost of indexing elements columnwise
COST∆LUP - cost of executing a loop

In addition, the following variables denote factors related to the file structure itself:

REC - total number of records
FLD - total number of fields
BLK - total number of blocks
REC∆BLK - number of records per block
FLD∆BLK - number of fields per block

Lastly, the following variables denote the factors related to the retrieval of records:

REC∆RET - number of records retrieved
FLD∆RET - number of fields retrieved
BLK∆RET - number of blocks retrieved
FLD∆LKP - number of fields searched

We can't explore every possible way that records can be retrieved. We will consider the most commonly occurring case: when records are retrieved in random order, without duplication, from an unordered data base [Note 3].

The first cost aspect we will compute is that of reading blocks. We can compute this cost by:

COST∆READ←COST∆BUF×⌈(REC∆BLK×FLD∆BLK)÷SIZE∆BUF

As one might expect, the cost of reading a block is determined by the size of the block.

The second cost aspect, that of executing loops, is discussed in the section of this paper which deals with algorithms. We will simply refer to it as *COST∆LUP*.

The third cost aspect we will compute is that of extracting records. We compute this cost by:

COST∆INDEX←(COST∆ROW×REC∆RET÷BLK∆RET)+COST∆COL×FLD∆RET

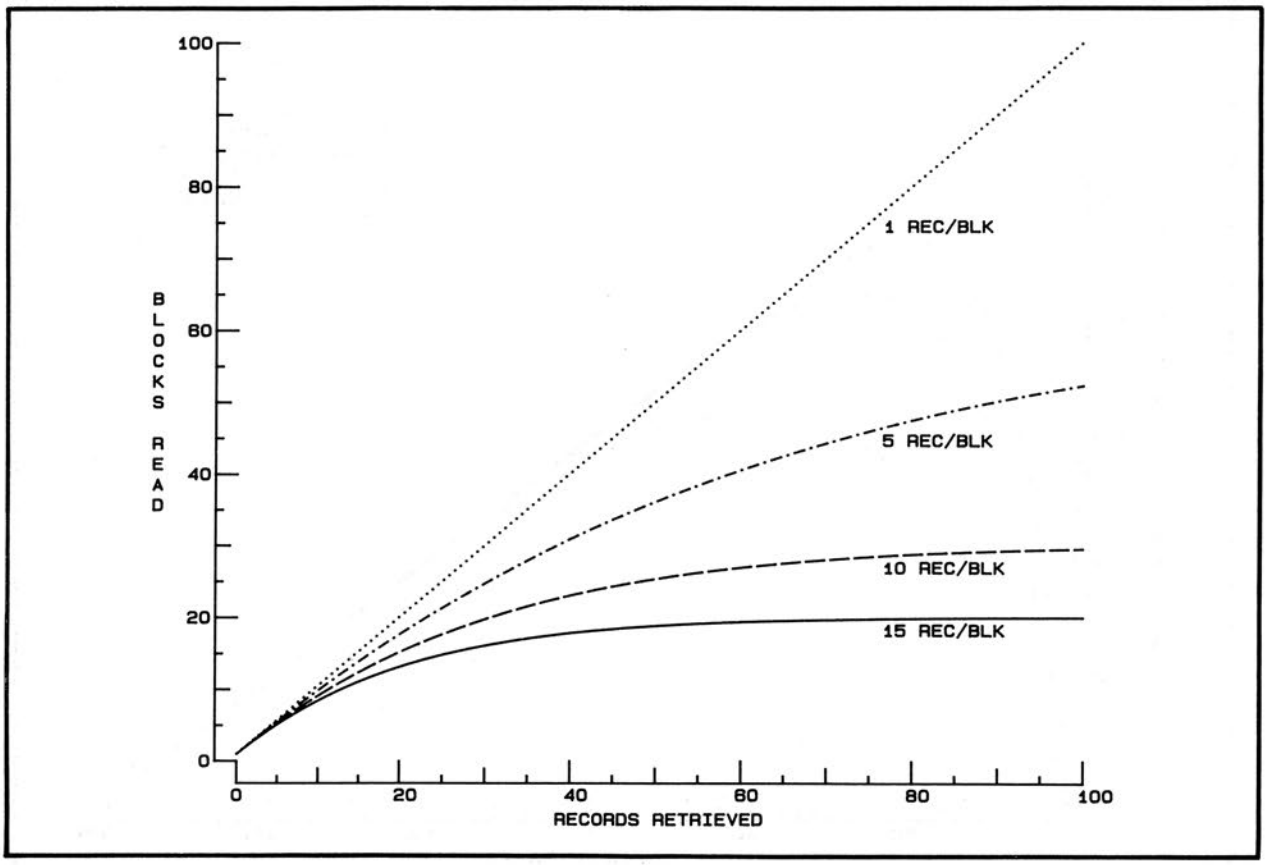
Putting everything together, we can compute the cost of reading a block and extracting records from it by:

COST∆BLK←COST∆LUP+COST∆READ+COST∆INDEX

Thus, the total cost of record retrieval is: *BLK∆RET×COST∆BLK*

Now that we know how to figure the cost of retrieval, let's look at an example. To show the effect of various configurations on cost, we will use a sample data base of 300 records and 100 fields . We will test the effects on performance when we choose block sizes of 1, 5, 10, and 15 records respectively. If we plot the expected number of blocks read, we get the following graph:

Figure 2



As the graph shows, as the number of records retrieved increases, so does the number of blocks read. This increase is rapid when we choose a smaller block size. If we increase the block size, we reduce the number of loops executed. This does, however, have an adverse effect on the cost of reading and extracting records.

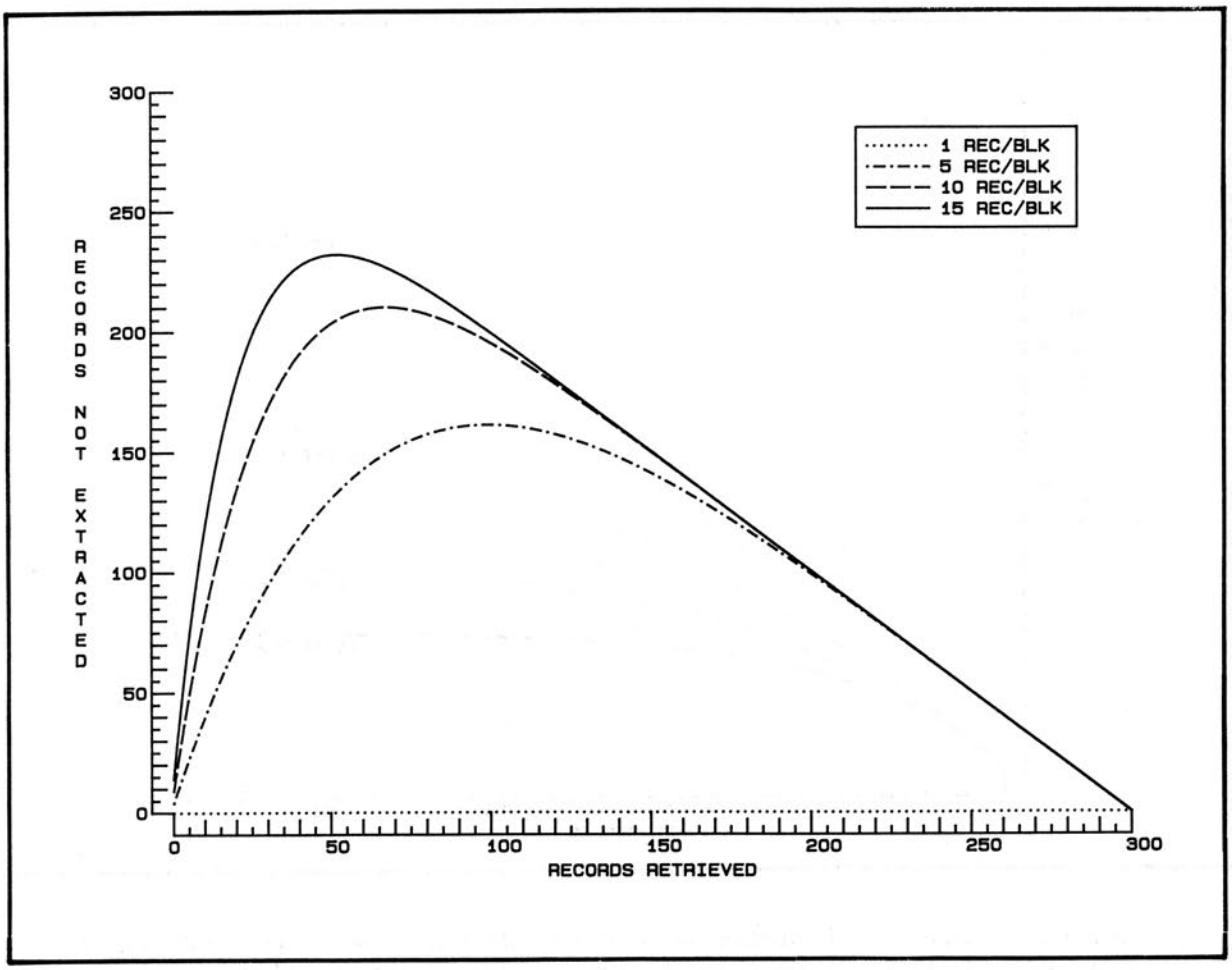
In the example above, the cost of reading a block is:

```
COSTΔBUF×⌈(100×1 5 10 15)÷SIZEΔBUF
COSTΔBUF×⌈(100×1 5 10 15)÷3156
COSTΔBUF×⌈100 500 1000 1500÷3156
COSTΔBUF×1 1 1 1
```

The change in block size has virtually no effect on the read cost in these cases. This is due to the small block size we have looked at. A block size of 50 records doubles the cost of reading a block.

The following graph shows the amount of extra data read for the varying block sizes and record retrievals, "EXTRA" data being records read into the workspace but not extracted.

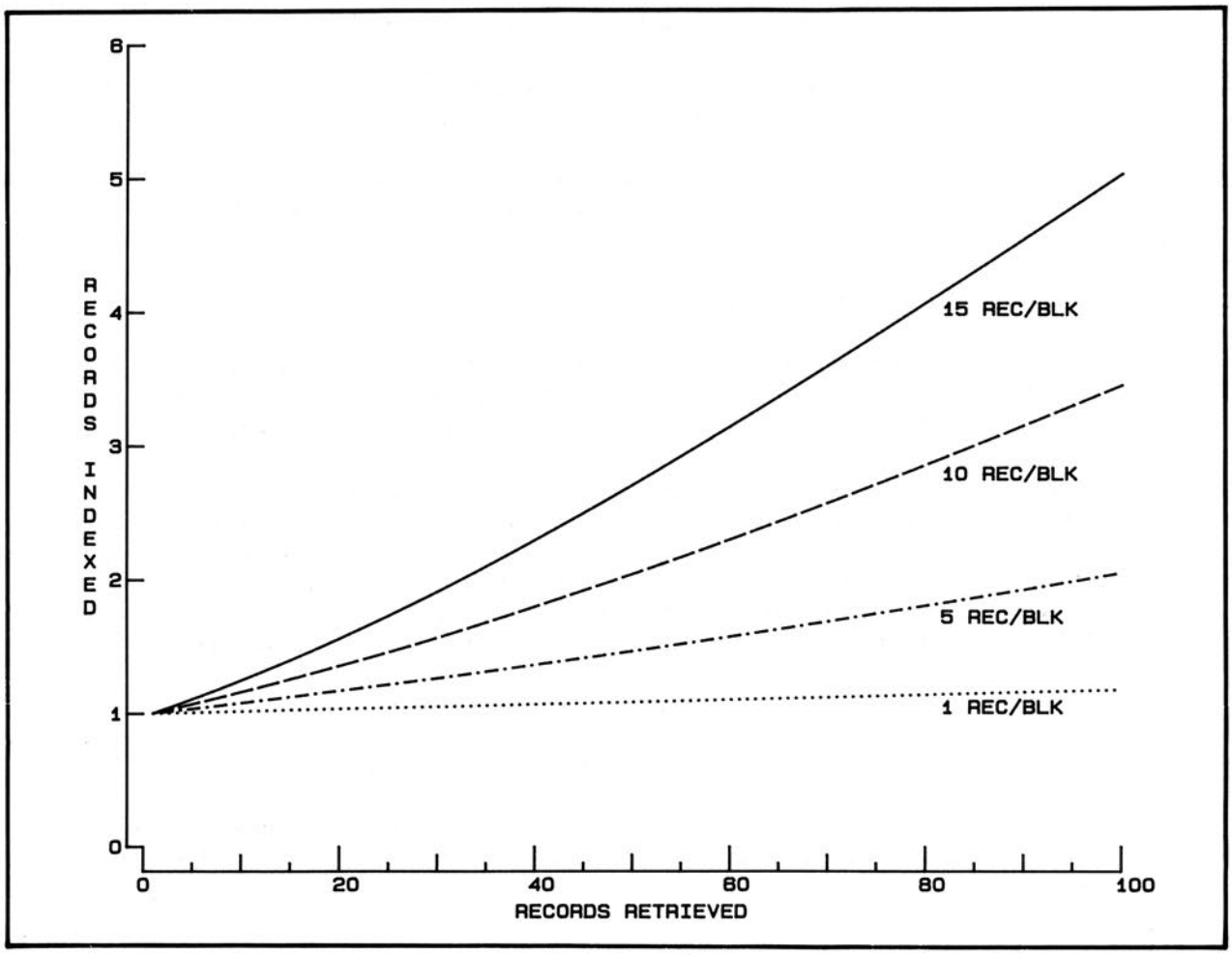
Figure 3



The graph shows that larger block sizes are more productive when used with larger record retrievals.

Since complete records are being read, field indexing cost is assumed to be constant. If we plot the mean number of records read per block, given the various block sizes in the example, we get the following graph:

Figure 4



The graph shows that the cost of indexing rises in proportion to the number of records read per block. Thus, it is sensitive to an increase in block size and the number of blocks read in any retrieval.

We can assume that the cost of indexing is relatively small compared with the cost of reading a block from file. Given this assumption, the actual per block cost of reading, looping, and indexing remains fairly constant regardless of block size. The major cost is determined by the number of times these operations are performed. It is directly proportional to the number of blocks retrieved. As shown in Figure 2, an increase in block size reduces the number of blocks which must be read. More operations, each of which is less costly, are performed in each retrieval. For small record retrievals, the difference in block size has little effect on cost.

We can draw the following conclusions from this analysis:

- If we know that our application will often retrieve large numbers of records, we will benefit from using a large block size.
- If we know that our application will often retrieve small numbers of records, we will benefit from using a small block size.
- If we are going to optimize the efficiency of an uninverted structure with regard to retrieval, we must know how the application will be used.

Searching - Uninverted Design

So far, we have explored uninverted files from the viewpoint of record retrieval. When we design a data base, we may also be concerned with searching. If the file is not sorted or in some special order, how will a program which searches the structure perform?

We must consider an additional cost aspect in searching. This factor, *COST∆LKP* represents the lookup costs per data element. Given the nature of the uninverted data design, a search requires looping through the entire data base, extracting the appropriate fields and performing the lookup. In the examples to follow, we assume that a search will be followed by retrieval. Since the block will already be in the workspace, no extra reading from the file is required. We can compute the total search cost by:

```
COST∆INDEX←(COST∆COL×FLD∆LKP)+COST∆ROW×REC∆BLK

COST∆BLK←COST∆INDEX+COST∆LKP×REC∆BLK×FLD∆LKP
```

The first equation computes the cost of extracting data from a field prior to looking it up in the block. For example, if one field were to be searched in a file blocked to 15 records, the cost would be:

```
(COST∆COL×1)+COST∆ROW×15
```

In addition to indexing, search cost is proportional to the number of elements searched. Thus, the per block cost of searching in this example would be:

```
COST∆BLK←COST∆COL+15×COST∆ROW+COST∆LKP
```

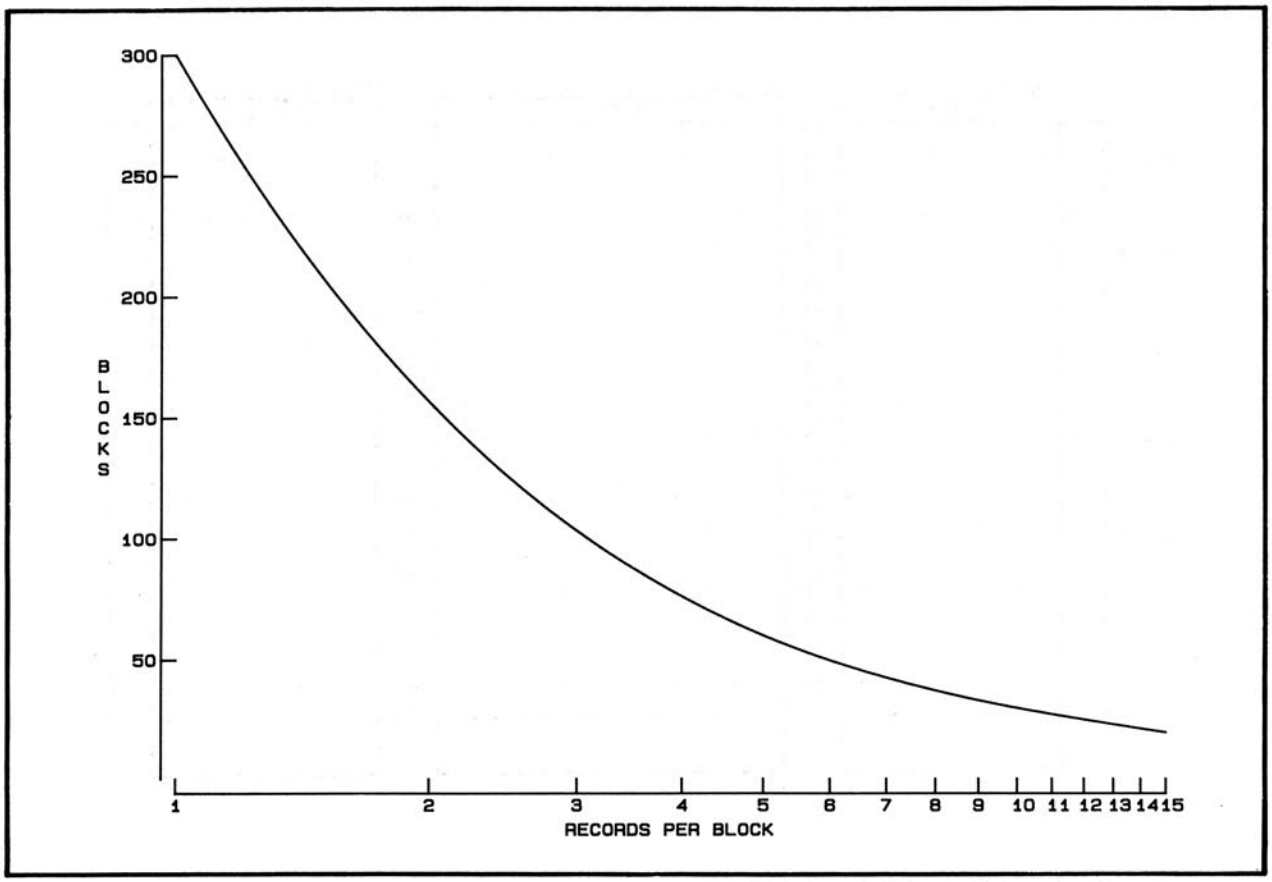
Since a search requires reading *all* the blocks, the total search cost is:

```
BLK×COST∆BLK
```

This search cost must be added to the cost of retrieval. In a one field search, *COST∆COL* remains constant across varying block sizes and will not affect the overall search cost. In this case, the total number of blocks in the file and size of the blocks will be the major factors of cost. Since these are inversely proportional, as shown in

Figure 5, we can conclude that a search is more efficient when performed over fewer blocks of more records. This is also true for record retrieval. Our final conclusion is that if we are going to use the uninverted structure we should use a large block size in order to minimize searching costs.

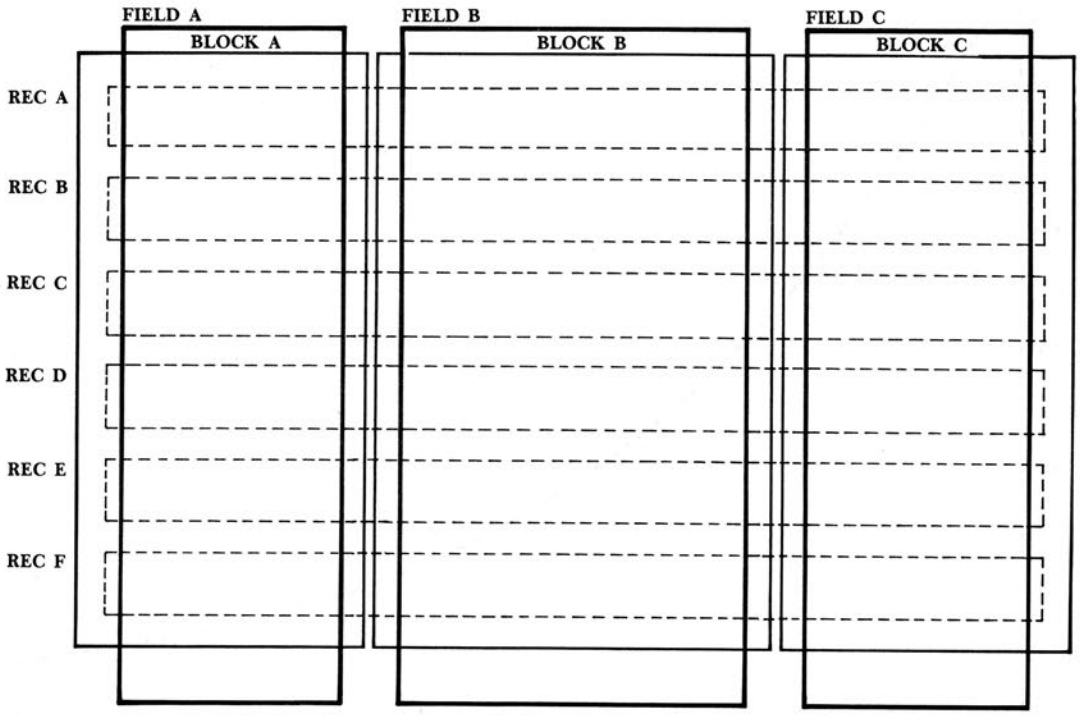
Figure 5



Inverted Data Design

The obvious alternative to an uninverted structure is an inverted one. We are using an inverted structure when we store two dimensional data (records by fields) in "field-major" order. Instead of grouping information because it belongs to the same record, it is grouped because it belongs to the same field. Figure 6 displays the physical layout of an inverted file structure.

Figure 6



Record Retrieval - Inverted Design

How does the inverted design compare with an uninverted one for both search and retrieval? As before, we will analyze the data structure in various circumstances.

Once again, we will consider the case when records are retrieved in random order without duplication from an unordered data base. We will assume that the total records per block is equal to the total records in the data base (*REC*Δ*BLK=REC*). The reason is that an inverted block contains all the records for a given field. Also, the total number of fields is equal to the total number of blocks (*FLD=BLK*). This is, of course, the basic nature of the inverted file structure.

Just as with the uninverted structure, there are three aspects of retrieval cost. These are the cost of reading blocks, executing loops, and extracting records.

We can compute record retrieval cost by:

```
COST∆READ←COST∆BUF×⌈REC÷SIZE∆BUF
```

Using the previous example of a 300 by 100 element data base, only one file system record is read from file (`⌈300÷SIZE∆BUF` ↔ `⌈300÷3156`).

We defer the question of looping costs to the next section.

Once accessed, the cost of indexing record elements is:

```
COST∆INDEX←COST∆ROW×REC∆RET
```

Since field data is stored as vectors, there is no columnwise indexing needed.

Putting it all together, we compute the per block cost of retrieving records by:

```
COST∆BLK←COST∆LUP+COST∆READ+COST∆INDEX
```

This process is repeated for each field retrieved (*FLD∆RET*). Thus, the total cost of record retrieval in the inverted data file is: *FLD∆RET*×*COST∆BLK*

Comparing the cost of record retrieval in an uninverted versus inverted structure, reveals an important difference between the two. Suppose a "complete" record is retrieved in both systems. For the uninverted file the retrieval cost is:

```
(1×BLK∆RET)×COST∆LUP+(1×COST∆BUF)+(1×COST∆ROW)+100×COST∆COL
```

With the inverted file, the cost is:

```
(100×FLD∆RET)×COST∆LUP+(1×COST∆BUF)+1×COST∆ROW
```

The only change is the shift from indexing 100 fields in the uninverted file to reading 100 blocks in the inverted file. Clearly the cost difference is significant. With a larger retrieval, would there be a major change in this relationship? From the two equations, it is apparent that an increase in the number of records retrieved has little effect on retrieval cost in the inverted file. This is not true for the uninverted file; cost rises with an increase in retrieval size. Importantly, larger retrievals with the inverted design, can use each access more productively, as the cost is similar to that of smaller retrievals.

Searching - Inverted Design

From the outset, the inverted design appears to be perfectly suited to field searching. After all, the only cost associated with a single field search is:

$$COST\Delta SEARCH \leftarrow COST\Delta LUP + COST\Delta READ + COST\Delta LKP \times REC$$

This is significantly faster than reading *all* the blocks as was necessary with the uninverted design. Therefore, the cost of searching only increases with the number of fields searched. The total cost for both searching and retrieval in an inverted design is:

$$(FLD\Delta RET \times COST\Delta INDEX) + (FLD\Delta RET + FLD\Delta LKP) \times COST\Delta SEARCH$$

As before, we assume that a search will be followed by a retrieval. Any fields read for searching will not be re-read for retrieval.

Since the record indexing cost remains relatively constant regardless of the number of matches found in a search, the majority of the cost lies with the number of fields retrieved. Thus, an inverted file is well suited for search and retrieval applications especially those requiring larger record accesses.

Data Structures and Files - Summary

There are many ways to structure data in a file. We have explored only two: uninverted and inverted structures. Each one has advantages over the other, depending on its configuration or usage. For example, uninverted data designs are best used in applications with large "complete" record retrievals; inverted ones are better for search and retrieval applications, especially those with larger record matches and accesses.

Another aspect of data design is its effect on performance in record updating. There are basically two types of updates: adding records or changing records. When we add a record to a data base we are essentially performing a complete record retrieval in reverse; thus many of the performance considerations are identical to those for retrieval.

When we change records already in the data base, we first locate and extract the appropriate records, make the changes, then replace the records. This is analagous to performing a search, a retrieval then a retrieval in reverse. Thus, the overall performance is determined by the costs associated with these steps.

An external data design must be tailored to the uses of an application. To be most effective, it needs to be adaptable to the changing needs of the users. A proposed data structure should be considered from several different angles, if for no other reason than to know which applications it is *not* suited for. By recognizing the components which comprise the cost of operation, a programmer can make effective decisions relating to the design itself, and how that design serves the application.

Program Structures and Files

When we design an APL application system, we usually devote the bulk of our development time to the overall logic of the system and the flow of data. On occasion we need to structure the flow of programs as well. Before the advent of the ⍎⍎*PACKAGE*⍎⍎ data type in SHARP APL, the typical external program structure stored the character representation of one function in each component of a file. Packages have made it possible to store several functions, or functions and variables, in a single file component. The immediate motivation for putting functions on file was usually to increase the space available in an active workspace. Stored functions also provided further advantages which added new dimensions to APL production systems. Some of these are:

- **Shared Functions**: It is possible to share functions between several systems, and make it unnecessary to update new versions of functions in several workspaces.
- **Local Functions**: It is possible to localize stored functions to the main calling function, thus maintaining a clean global environment.
- **Enhanced Security**: Local stored functions provide further guarantees against program tampering.
- **Audit Trail Information**: It is possible to use the file system's records of who updated the stored function and when it was done.

As you can see, the benefits of storing functions on file are similar to those of storing data on file.

Many people have written facilities to make storing, changing, and using externally stored functions easier. These are usually tailored to the particular application. We will look at paging in more general terms. We will discuss two types of schemes: process paging and demand paging. The term "paging" comes from the technology of virtual memory systems. Such systems move pages of virtual memory in and out of real memory as they are needed. The analogy with APL workspaces and files is obvious.

Process Paging

Process paging is similar to the way we use data which is stored on files. Data specific to the current process is read from file, manipulated and expunged before going on to the next process. For example, a system which produces a monthly sales report might use the following function for printing.

```
    ∇ MSREPORT MY;DATA
[1]  →(DATE∆SALES MY)ρOK ◇ 'INVALID DATE' ◇ →0
[2] OK:DATA←MY FREAD 'SALES'
[3]  MY PRINT∆SALES DATA
    ∇
```

While not a particularly exciting function, it does illustrate the similarities between functions and data which are related to specific processes. Sales data is read in from file (via the function *FREAD*) and is used to produce the monthly report. When the report is complete the data is expunged. We assume the two functions *DATE∆SALES* and *PRINT∆SALES* have been saved in the workspace along with *MSREPORT*. As with the data, there is no reason why these two functions need to permanently reside in a workspace. In fact, it is possible that one or both might be useful in other systems as well. If a change is needed, a programmer propagates the updated function into all the workspaces requiring the change. It is convenient to store these functions externally and bring them in only when a process calls for them specifically. Thus, sub-functions and local data remain in the workspace for only as long as they are needed. With this in mind, the function can be changed to the following:

```
    ∇ MSREPORT MY;DATA;DATE∆SALES;PRINT∆SALES
[1]  PAGEIN 'DATE∆SALES PRINT∆SALES'
[2]  →(DATE∆SALES MY)ρOK ◇ 'INVALID DATE' ◇ →0
[3] OK:DATA←MY FREAD 'SALES'
[4]  MY PRINT∆SALES DATA
    ∇
```

We can go one step further. We can write a **cover** function which does the housekeeping for the report. For example:

```
    ∇ MSREPORT MY;MSREPORT;DATE∆SALES;PRINT∆SALES
[1]  PAGEIN 'MSREPORT DATE∆SALES PRINT∆SALES'
[2]  MSREPORT MY
    ∇
```

Now the copy of *MSREPORT* on file contains only the code designed to run the report, as in the first example. The workspace-resident copy does only page-related operations. This has several advantages. It is possible to change a function even when a user has saved a private workspace which contains that function. Also, the version in the workspace takes very little space when it is not executing.

Suppose that a change is to be made to the report and a new sub-function must be added. Since these functions have been specifically localized in the workspace version and explicitly paged in, the workspace-resident functions will no longer work as intended. The new sub-function, which is not localized, will remain in the workspace.

Therefore, we want to page in groups of objects without localizing them and either expunge them when the process is complete or when more workspace is needed. In this example, the group could be "SALES". With some minor coding, the function *PAGEIN* could be modified to handle groups of objects. The *MSREPORT* could now be written as:

```
    ∇ MSREPORT MY;MSREPORT
[1]  PAGEIN 'SALES' ◇ MSREPORT MY
    ∇
```

What is the effect of process paging on performance? Obviously this depends on how the page file is designed. But regardless of design, there is an increase in cost due to added file operations. If the report is run infrequently this increase is probably recovered by the benefits gained. But in situations where it is run frequently, this method adds significant cost to its execution. In the latter case a preferred method is to page objects in only when they are needed, without the use of a cover function, and remove them when we need the space. This method of paging is called **demand paging**.

Demand Paging

Prior to the advent of event trapping in APL systems [Note 4], demand paging was similar to process paging. The difference was that the demand could not be handled dynamically. For example, a user sitting at a terminal can bring functions and variables as they are needed by copying groups.

```
      )COPY SALES REPFNS
SAVED ......
      )GRP REPFNS
MSREPORT DATE∆SALES PRINT∆SALES
```

When finished,)*ERASE REPFNS* is all that is needed to clean up the workspace.

The same effect can be achieved using process paging with a simple grouping scheme. For example:

```
      PAGEIN 'SALES'
```

brings in from file the functions for the sales report. This is analagous to the *COPY* command, but can be done under program control. After execution the application must expunge the objects.

With demand paging we eliminate the need to explicitly read and explicitly expunge the objects, we simply execute them. This is accomplished through the use of event trapping. Event trapping allows a programmer to indicate a specific action to be taken when a trappable event occurs. The action can be a paging operation.

Recently a new type of event handling was introduced in SHARP APL which makes it easier to do demand paging. In most event trapping, an error occurs and control is passed back to the application for action. When the necessary recovery actions have been completed, the application restarts the stopped function by doing →⎕*LC* or something similar. The line which stopped in mid course restarts at the beginning. But for a demand paging situation the event handler needs to return control at the point of the error wherever it occured in the line. This can be done using an *immediate* trap.

In the example of the monthly sales report, the key entry point into the process is the execution of *MSREPORT*. As has been shown, there are a number of ways the subfunctions and data can be paged in. In a demand paging situation the ideal method is to take notice of the use of this key function and minimize the amount of work needed to prepare for its execution.

Suppose a workspace contained a global non-default ⎕*TRAP*; defined as:
⎕*TRAP*←'∘ 6 *I PAGEIN* ⎕*ER*'. This definition can be explained by the following:

> "Upon encountering a *VALUE ERROR* (event number 6), execute the expression *PAGEIN* ⎕*ER immediately*; at the point of the error midline. If the error is resolved, continue processing the original line; otherwise halt execution and signal the original error".

Processing in this workspace proceeds in the following manner:

```
      ⎕NC 'MSREPORT'
0                          The object is not defined currently.

      MSREPORT 9 1982      A VALUE ERROR occurs, but is immediately trapped.
                           The last row of ⎕ER reports the name which was not
                           defined. Thus, PAGEIN 'MSREPORT' was executed, the
                           object defined, and execution proceeded without inter-
                           ruption.
```

The expense of bringing in the functions is incurred this first time. On subsequent calls to *MSREPORT* no *VALUE ERROR* will occur, and the report executes without any additional overhead.

The function *PAGEIN* needs to perform certain housekeeping duties on occasion. For example, it is desirable to limit the amount of work space consumed by paged-in functions not currently in use. Thus, *PAGEIN* could check some threshold to see if it can fit in a new set of functions. If not, it determines which functions to expunge possibly on a first-in-first-out basis.

The major difference between the demand paging and process paging is that demand paging is implicitly invoked by an event trap. Because demand paging works on an "as needed" basis, it will usually do less paging and thus be more efficient. If you must store functions on file, and the necessary facilities are available, demand paging is the way to go.

Program File Design

The performance of any paging method is dependent on the structure of the function file. The basic aim of any function file design is to minimize the number of file operations required to fetch a group of functions. It should also minimize the number of objects brought in. Storing the character representation of each function in a separate component can degrade performance even when paging a single function into the workspace.

The SHARP APL **package** data type provides the capability to store different APL objects in a variable. In particular, a single package variable could contain a collection that includes both functions and variables (including other packages, and numeric, character, and enclosed arrays). To use objects stored within a package, they must be extracted or defined. Since multiple objects can be stored in a package, designing a function file which uses packages can reduce file operations.

If an application has several main options, each option can be stored and run in its own environment. The program file can contain a **library** of these packaged options, each stored in a separate component. Therefore, each package contains all the functions and variables needed to run the option and is similar to a stored workspace. Storing an entire workspace in a package variable reduces the number of file operations required to bring objects in. Unfortunately, it also increases the number of objects retrieved that are not needed.

Suppose a workspace contained only the function *MASTER* and had `⎕LX←'MASTER'`. When the workspace is loaded the function *MASTER* does the following:

- Ties the file containing all the packaged objects.
- Asks the user for the desired option.
- Defines the package from the file.
- Runs the rest of the option *within* the packaged environment [Note 5].

When the option completes, the associated objects are implicitly expunged because they are localized. The user is now back to a clean environment, ready to begin a new option. If all the objects needed for a given task are stored together, only one file operation is needed to read them in. More file operations are required to do several options than with the whole workspace approach, but fewer extraneous objects will be brought in.

Segmenting options into packaged groups reduces the number of file operations. It is important to ensure that these groups are divided finely enough so as to minimize the number of unneeded objects brought in. It is also important to reduce the amount of duplication across groups. For example, in a large group scheme such as a packaged workspace structure, we might keep the same function stored within many of the groups. The function file should be designed to keep these duplications to a minimum.

Thus, the function file should be a collection of specific groupings of objects. Unfortunately, this adds a number of file operations to the process, but each operation involves moving less data and is competitive with the large group setup.

Regardless of group size, it is important to avoid any complicated directory structure which increases the overhead for locating a specific group. This is why a "hashed" file approach is attractive. By **hashing**, we mean an algorithm which maps the name of the function into a component number. Typically, hashing schemes map a large number of possible names into a relatively small number of locations. If the distribution of names were even when stored in alphabetical order, the application program could find a name without any directory lookup at all. But typically this is not the case. By using the actual number of locations in the file as a parameter in hashing names into locations, groups can be more evenly distributed.

The design of the function file, as with data files, depends on the expected uses of the application. If the division of the objects on the file is very fine, the number of file operations to locate an object will be lower. The possibility of repeatedly going to the function file for a new object is increased. If division of the objects on the file is very coarse, unneeded objects will be read from file, but the number of file operations will be lower. When using logical groupings, other than in the case of the packaged workspace concept, there is additional overhead for object location. It may even take several file operations to actually locate the object. But given a concise grouping the number of new object pages will be minimal.

The trade-off between retrieving unneeded functions and doing repeated file operations has an analogy. The same trade-off occurs between retrieving unneeded records and doing repeated file operations, as was discussed in the section on data structures. The designer must choose where to strike a balance based upon his knowledge of how the application will be used.

Most novice APL programmers attempt to improve the efficiency of an application after it is running by looking for a faster primitive function. Experience shows, however, that the greatest gains are made by improving the data and program file structures. If you consider the issues discussed above as you design an application, you will design systems which are efficient from the start.

ALGORITHMS

Solutions in APL, as with other programming languages, can be expressed in many ways; though each way has advantages and disadvantages. The "best" solution is one which performs best in the particular situation in which it will be used.

We should consider several factors when choosing an algorithm, such as:

- work space availability
- space requirements of primitive functions
- argument type and shape
- context of usage

These factors can influence the performance of APL functions; both positively and negatively. Thus it is important to be aware of all areas which affect an algorithm's efficiency.

The following section deals with algorithms as they relate to "flow control" in APL functions.

Branching

Branch statements are used to redirect the sequence of execution within a function. The expression which follows a branch arrow is one of the following:

- a valid line number
- an invalid line number
- empty vector
- nothing

There are many ways to calculate a branch destination. Some of the more common forms are:

If we assume that `(1≥ρρ`*TEST*`)∧1=×/ρ`*TEST*, we can do:

`→`*TEST*`ρ`*LABEL*
⍝ Branch to *LABEL* when *TEST*`>0`. For *TEST*`=0`, continue with the next statement.

`→`*TEST*`↓`*LABEL*
⍝ Branch to *LABEL* when *TEST*`=0`. Otherwise continue with next statement.

→LABELS[TEST]
⍝ Branch to *LABELS[TEST]* when *TEST* is within the range of *⎕IO* to *⍴LABELS*.

→TEST⍎LABELS
⍝ Branch to *LABELS[⎕IO+(⍴LABELS)|TEST]* .

→TEST↓LABELS
⍝ Branch to *LABELS[TEST+⍳TEST<⍴LABELS]* when *TEST≥⎕IO*.
When *TEST<⎕IO*, branch to *LABELS[⍳(|TEST)<⍴LABELS]*.

If we assume that *(1=⍴⍴TEST)∧1≤⌈/⍴TEST*, we can do:

→TEST/LABEL
⍝ Branch to *LABEL* if *∨/TEST>0*; continue if *∧/TEST=0*.

→TEST/LABELS
⍝ Branch to *LABELS[TEST/⍳⍴TEST]* if *∨/TEST>0*; continue if *∧/TEST=0*.

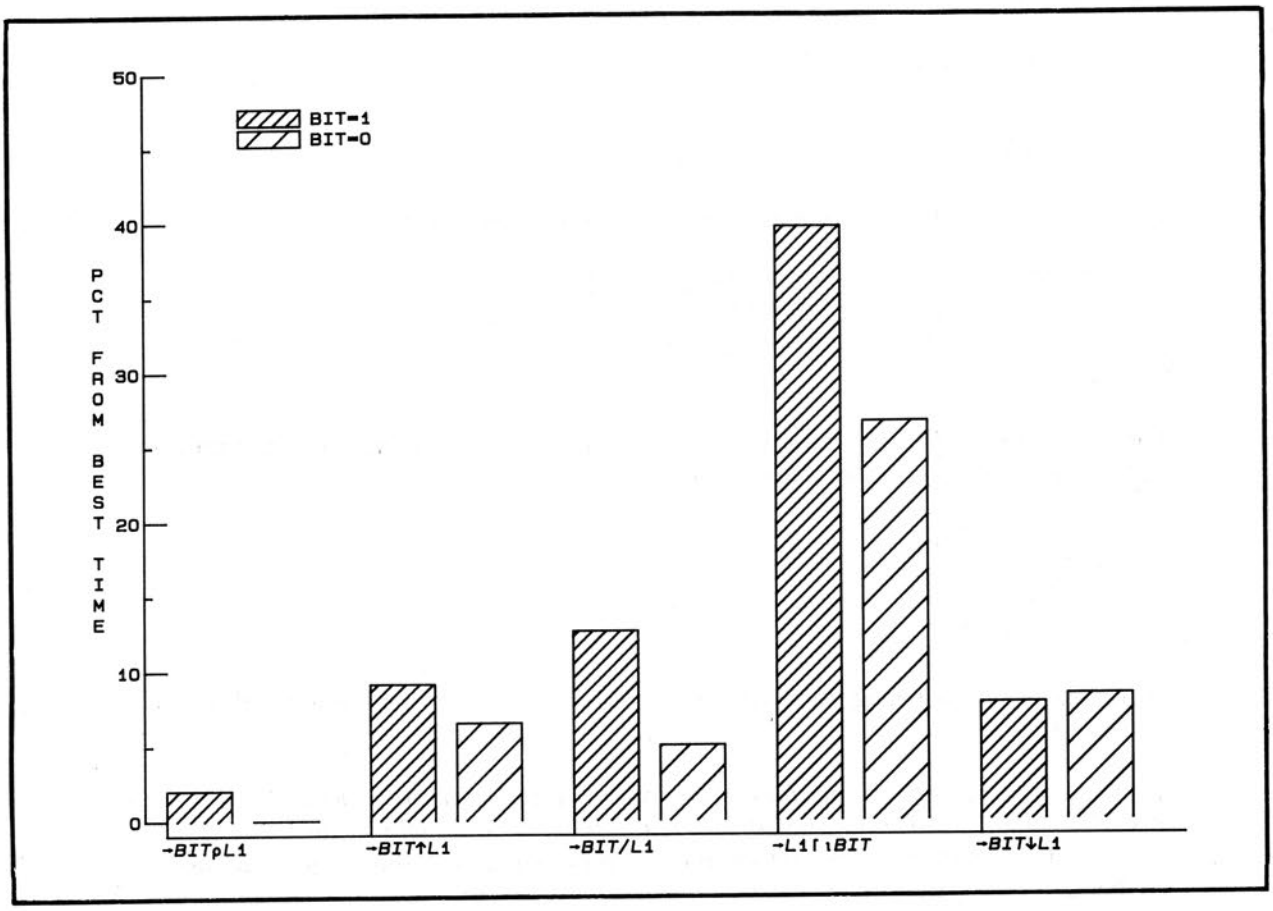
Computed branching in APL functions provides a powerful means of directing order of execution. It allows functions to respond to parameters at run-time and eliminates repetitive segments of code.

The following graph shows some common forms of branching and their relative timings in the current version of SHARP APL. Timings such as these can be useful in choosing a style of branching which best fits an application.

As the graph shows, the condition which controls the branch can vary the execution times. For example, *→L1⌈⍳BIT* is faster to execute when *BIT←0* than when *BIT←1*. Therefore, branches should be well thought out when trying to make an application as efficient as possible.

Figure 7

Effect Of Branch Condition
On Relative Execution Time



A common use of branching in APL applications is for looping, as shown in the following:

```
[∘]  LIMIT←⍴DATA ◇ COUNT←0
[∘] LOOP: →(LIMIT<COUNT←COUNT+1)⍴END
[∘] PROCESS DATA[COUNT] ◇ →LOOP
[∘] END: ...
```

In the example above, the parameters *LIMIT* and *COUNT* are set as early in the function as possible; preceding the point where the parameters are checked. Iverson, in *A Programming Language*, described this as the *Method of Leading Decisions*. This

allows a function to behave properly even when unusual values for the parameters are encountered; for example when 0=ρ*DATA*. The following table illustrates the major segments of the looping process and the number of times each segment executes when the shape of *DATA* is 10:

1) *LIMIT<COUNT*	11
2) →ι0	10
3) →*END*	1
4) →*LOOP*	10

We can also loop by checking parameters *after* the process, as in:

```
[∘]  COUNT←1 ◇ →(COUNT≤LIMIT←ρDATA)↓END
[∘] LOOP: PROCESS DATA[COUNT]
[∘]  →(LIMIT≥COUNT←COUNT+1)ρLOOP
[∘] END: ...
```

The following table illustrates this looping process and the number of times each segment executes when the shape of data is 10:

1) *LIMIT≥COUNT*	11
2) →ι0	2
3) →*END*	0
4) →*LOOP*	10

As we can see, switching to a trailing check has reduced the number of branches performed.

In both a leading and trailing decision loop, we perform an *end-of-loop* check through each pass of the loop, even though we know that we will only branch out once. To eliminate this check, we can precompute the condition in advance. Both functions can be changed to reflect this, as follows:

For leading decisions:

```
[∘]  CHECK←((ρDATA)ρ0),1 ◇ COUNT←0
[∘] LOOP:→CHECK[COUNT←COUNT+1]ρEND
[∘]  PROCESS DATA[COUNT] ◇ →LOOP
[∘] END: ...
```

For trailing decisions:

```
[∘]  CHECK←((ρDATA)ρ1),0 ◇ →CHECK[COUNT←1]↓END
[∘] LOOP: PROCESS DATA[COUNT]
[∘] →CHECK[COUNT←COUNT+1]ρLOOP
[∘] END: ...
```

We can illustrate both these loops with the same shape data as before with the following result:

For leading decisions:

```
1) LIMIT<COUNT          0
2) →⍳0                 10
3) →END                 1
4) →LOOP               10
5) CHECK[COUNT]        11
```

For trailing decisions:

```
1) LIMIT≥COUNT          0
2) →⍳0                  2
3) →END                 0
4) →LOOP               10
5) CHECK[COUNT]        11
```

Obviously, all extraneous work done inside a loop is costly. For one thing, it means the work is executed each time through the loop, often unnecessarily. Therefore, our main goal is to keep the loop down to its most essential segments. These segments are:

- Initialize paramaters
- Do process
- Increment counter
- Check for end of loop

We can extend the idea of precomputing conditions by preassigning the branch destinations. Thus, we can direct the flow of the loop with minimal extraneous work. A trailing decision loop is better suited to the use of precomputed destinations and can be re-written to the following:

```
[⍳] →(CHECK←((⍴DATA)⍴LOOP),END)[COUNT←1]
[⍳] LOOP: PROCESS DATA[COUNT] ⋄ →CHECK[COUNT←COUNT+1]
[⍳] END: ...
```

We can analyze this process as before with the following result:

```
1) LIMIT<COUNT          0
2) →⍳0                  0
3) →END                 1
4) →LOOP               10
5) CHECK[COUNT]        11
```

As the table shows, this method of looping performs the loop with a minimum of the extra work associated with the looping process itself.

The fastest method though, is to change the process to handle all the data without a loop, such as in:

```
[∘]  PROCESS DATA
```

Execute

While discussing branching, it is appropriate to consider execute (⍎) as well. People often use it to control flow. Typical uses take the following form:

```
      ⍎(PARMS[1]=80)/'PROCESS DATA[1;]'
```

In this example of *⍎CONDITION/EXPRESSION*, there are two possibilities for execution. When *CONDITION* is true, *⍎EXPRESSION*; when false, ⍎''. Since the latter has no effect and no result, execution would proceed with the next statement. With the former, the APL system treats the character expression as it would immediate execution input. This involves overhead in building the internal representation of the string, among other things. This cost is *not* incurred if the same expression is rewritten using some other primitive, such as branching. If this occurs frequently, that is, if *CONDITION* is almost always 1, then extra work is repeatedly performed. In addition, there is the possibility of introducing errors such as *SYMBOL TABLE FULL*. If *CONDITION* is almost always 0, usage of execute may be warranted. Thus, it is important to consider the circumstances where execute is being used. On systems with diamond (◇), an execute of the above form can be expressed in the following way:

```
      →(PARMS[1]=80)↓L1 ◇ PROCESS DATA[1;]
L1: ...
```

Another frequent use of execute occurs when setting initial values for missing variables, as in the following:

```
PARM←⍎¯1 ↓ [⎕IO+0=⎕NC 'PARM']↓'PARM 1'
```

This is functionally equivalent to:

```
      →(0≠⎕NC 'PARM')⍴L1 ◇ PARM←1
L1: ...
```

Execute is a powerful function for building code dynamically. Occasionally this leads to uses which are better performed by other APL functions, as follows:

```
ORDER←⍎'⍋⍒' [⎕IO+COND],'ARRAY'
```

This can also be done without execute by:

```
ORDER←⍋ARRAY×¯1*COND
```

Finally, execute is often used as a form of multi-way branching, as in:

```
→⍎'L',⍕CHOICE          ↔      →(L1,L2,...)[CHOICE]
⍎OPTIONS[''⍴CHOICE;]   ↔      →(L1,L2,...)[CHOICE]
```

Usually, these are used in applications where new options need to be added quickly and easily. Occasionally, it is a temporary construct used in a new application to facilitate the growth of the system. Unfortunately, we all know what happens to temporary constructs; they become permanent.

Modularity

The need to perform a certain task may occur many times in an application. We can write a general purpose function to perform this task, to be used as a plug-in module where needed. If the extra cost of using a sub-function is relatively small compared to the overall application cost, and the code is repeated often, the segment is a good candidate for being placed in a separate module. If the code is trivial then it is probably better to repeat it, unless there is some overriding reason to avoid doing so. A case where it's useful to have a separate function, even though the task is trivial, is the often used function shown below:

```
    ∇ DEST←LABEL IF CONDITION
[1]  DEST←CONDITION/LABEL
    ∇
```

This function is useful for increasing readability in an application; with only minor degradation in performance.

Modular sub-funtions have some disadvantages; they tend to force programmers into certain styles and patterns. A problem does not always lend itself to such general-purpose solutions. For example, suppose we wish to express the following idea:

> "Branch to *SCA* when *DATA* is scalar; branch to *VEC* when *DATA* is a vector; otherwise continue."

If we were using an *IF* function we might code the above as:

```
→(SCA,VEC) IF 0 1=⍴⍴DATA
```

Alternatively, we might have coded this without *IF* as:

```
→(⍴⍴DATA)↓SCA,VEC
```

Over-modularizing can sometimes shield a programmer from more direct solutions to a problem. This also occurs when using "utility" functions. For example, we have all used functions which convert a vector to a matrix; either as defined functions or those native to APL. Quite often, one can find the following in an application:

```
MONTHS←'/' VECTOMAT 'JAN/FEB/MAR/APR/MAY/JUN/JUL/AUG/SEP/OCT/NOV/DEC'
```

as opposed to:

```
MONTHS←12 3⍴'JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC'
```

The months of the year do not change often. Since the lengths of the names are all the same, the reshape solution is more direct, more efficient, and almost as readable.

The purpose of this section has been to show some specific examples of how to choose efficient primitives and algorithms. Since the specifics vary from implementation to implementation, and even between releases of the same implementation, it is important to learn the general principles for choosing algorithms which are efficient. The best choice is normally the expression which is the simplest, most direct, and most specific to the problem at hand.

MEASUREMENT

When choosing a specific algorithm to perform a task, it is important to evaluate it under varying conditions of execution. There are several ways of doing this: at a *general* level, as in monitoring day-to-day resources for running the application; and at a *detailed* level, by timing the algorithms involved. The latter method will be discussed in this section.

Frequently, people confuse algorithm timing and benchmarking. A **benchmark** is an attempt at an exact determination of the run costs for a given algorithm. The results are often used to measure performance between APL systems or on different machines, whereas **algorithm timing** is a means of comparing the relative run costs within the context of an application. For the most part, the exact number of CPU units used by an algorithm is difficult to determine: it is dependent upon workspace available (⎕*WA*), argument size, user load, CPU load, etc. Therefore, the best approach is to time algorithms relatively; that is, in comparison to each other. This way any bias which may be present affects all the algorithms equally.

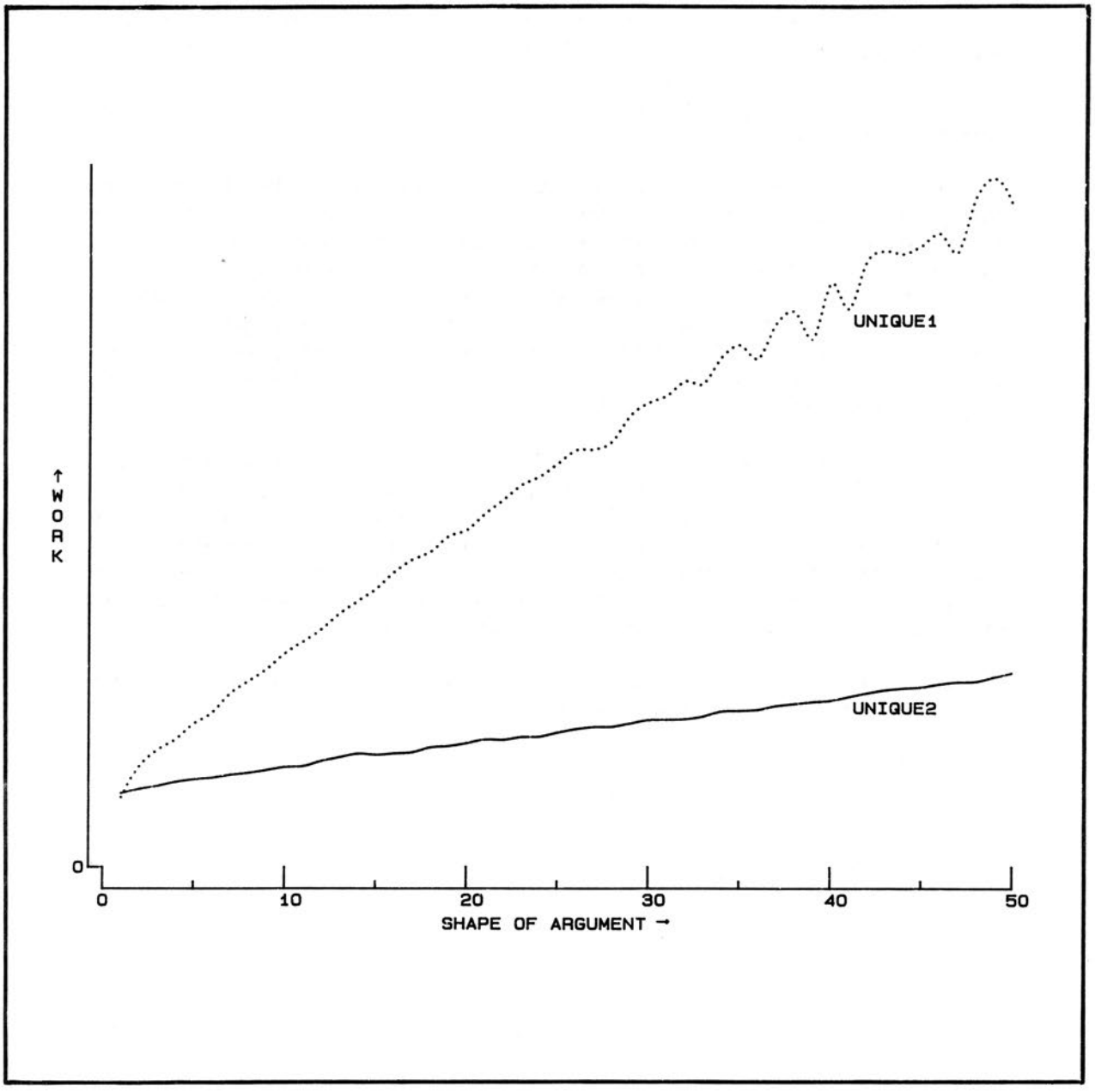
If we need an algorithm to select the first unique elements in a vector, what is the best choice? The first question to ask is, How long will the typical argument be? This is important because algorithms which are fast for small vectors may be slow for large ones. So an algorithm needs to be tested for speed of execution on several different argument sizes. Other questions which should be considered are: "How much space is available during execution?" or "What is the data type of the argument?". All these things are important when comparing the relative timings.

If, for example, the following two algorithms are chosen for comparison. The argument to be processed is in sorted order prior to execution:

```
UNIQUE1◇ R←((⍳⍴VEC)=VEC⍳VEC)/VEC
UNIQUE2◇ R←(¯1↓1,VEC≠1⌽VEC)/VEC
```

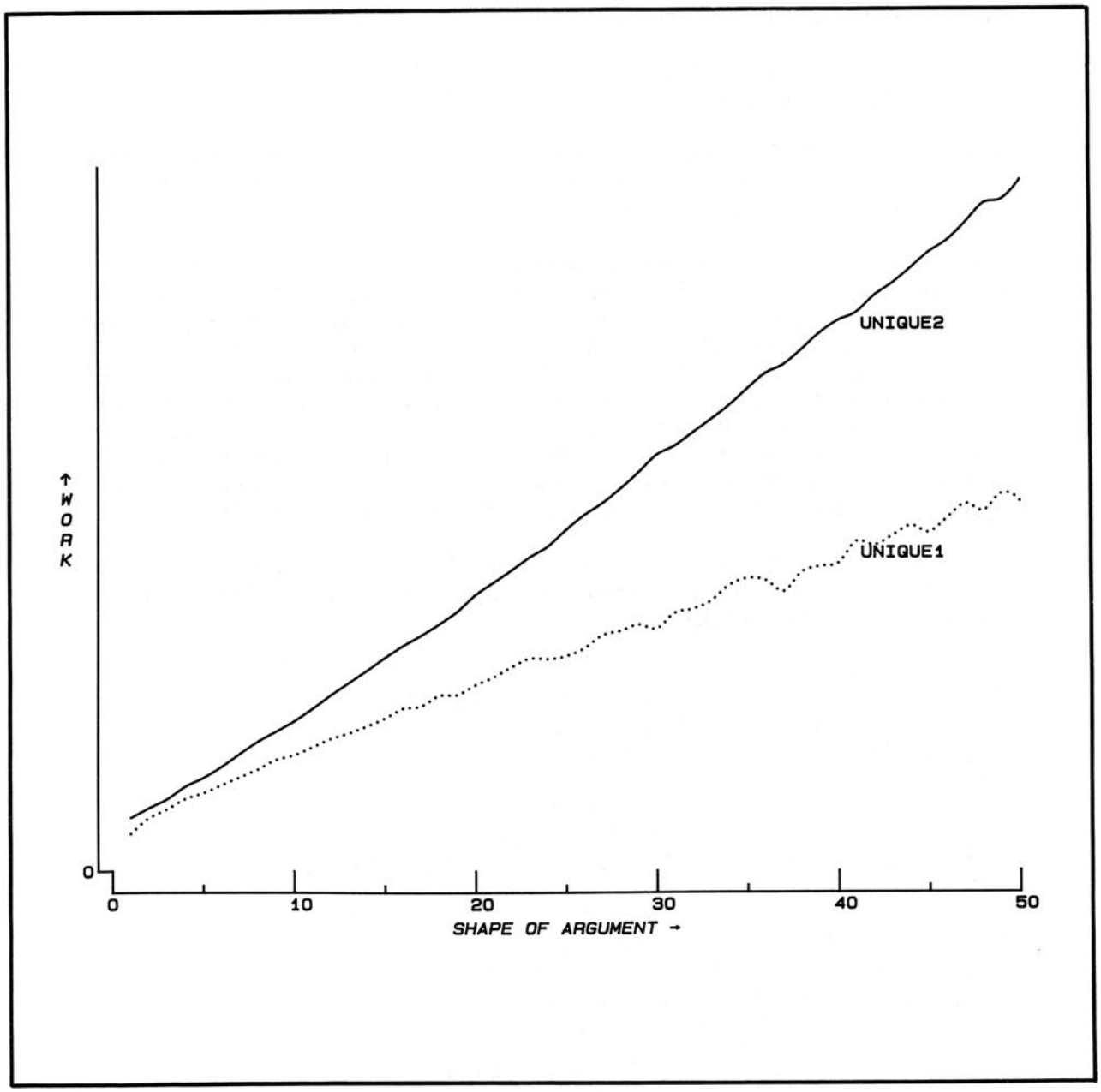
Using several different argument sizes the functions can now be relatively timed. Not surprisingly, *UNIQUE2* is faster for almost all size *VEC* used, as shown in the table below:

Figure 8



The reason for the difference is the order of the data within argument. The first function is general enough to work correctly on any ordering of the data. Since the data is in sorted order, there is no reason to perform a *dyadic iota* to find the first occurrences. The second function is specific to a sorted data argument, and spurious results are obtained when used with an unordered vector. Timing the two functions, including the cost of sorting in the second function, produces the following table:

Figure 9



The second function is relatively more expensive than the first. In addition, the results of the functions contain different orderings of the data when processed with a previously unsorted vector. Thus, it is important to be familiar with the context of usage to be sure the processes are identical in function. With a full knowledge of the data, its attributes can be exploited effectively.

The way to calculate computer resources used varies from system to system. The basic element of cost can be computed by the following, or variations of the following:

```
⍝ ⎕IO←1 ◇ ⎕AI[2]=SESSION CPU IN MILLI-CPU'S ON SHARP APL
TIME←⎕AI[2] ◇ ALGORITHM ◇ TIME←⎕AI[2]-TIME
```

Regardless of the method used, it is important to minimize any extra overhead between the calls to *⎕AI*. Furthermore, any overhead that is present should be considered constant regardless of the algorithm being timed.

The process of relative timing involves executing the algorithm for a sufficient number of times to generate a resonable statistical average. It is equally important to evaluate the variation between iterations of the same algorithm. In addition, the shape of the argument should be changed after each set of timings. This shows the effect of argument size on the execution of the function. Other external factors, such as system load, also have some effect; therefore, timings should be made at varying times of the day and, most importantly, at the time the application is normally executed. Timings made when CPU load is light need to be checked when the load is high. For the most part though, if functions are compared relative to each other, the relationship should stay approximately the same assuming the same bias is affecting each equally.

The following function is useful for timing single algorithms or systems with *⎕AI* or equivalent. Its result is the total, mean, and standard deviation, in milli-CPU for N repetitions of the specified algorithm:

```
    ∇ R←A CPUTIME B;C;CPUTIME
[1]  ⍝ COMPUTE THE APPROXIMATE MILLI-CPU USED FOR
[2]  ⍝ ALGORITHM <B> FOR <A> REPETITIONS. OVERHEAD FOR
[3]  ⍝ USAGE OF ⎕AI NOT CONSIDERED.
[4]  ⍝
[5]  ⍝ CONSTRUCT LOCAL COVER FUNCTION
[6]   C←⍴R←'CPUTIME←CPUTIME;A;B;C;R;A;B'
[7]   C←C⌈⍴B←'CPUTIME←⎕AI ◇ ',B,' ◇ CPUTIME←(-CPUTIME)+⎕AI'
[8]   B←(C↑R),[⎕IO-0.5] C↑B
[9]  ⍝ FIX LOCAL FN; PASS CONTROL TO SEQUENCE VECTOR
[10]  →(⍴⍴⎕FX B)⍴(R←(A⍴A),B)[C←⎕IO]
[11] ⍝ ⎕FX UNSUCCESSFUL; EXIT FUNCTION
[12]  'UNABLE TO BUILD LOCAL FUNCTION' ◇ →R← 0 0 0
[13] ⍝ EXECUTE ALGORITHM <B>, <A> TIMES
[14] A:R[C]←CPUTIME[⎕IO+1] ◇ →R[C←C+1]
[15] ⍝ DONE; CALCULATE TOTAL, MEAN ∈ STDEV
[16] B:R←B,C,((+/(R-C←(B←+/R←¯1↓R)÷1⌈A)*2)÷1⌈A-1)*0.5
    ∇
```

This is the basic function used for relative timing. The function, *RELTIME*, as follows, takes a list of algorithms and displays their relative times:

```
    ∇ R←RELTIME RELTIME A;B;C
[1]  ⍝ TABLE THE RELATIVE TIMINGS FOR <RELTIME>
[2]  ⍝ REPETITIONS OF ALGORITHM(S) IN MATRIX <A>.
[3]  ⍝ USES THE FUNCTION <CPUTIME> TO CALCULATE APPROXIMATE
[4]  ⍝ CPU FOR EACH ALGORITHM SPECIFIED.
[5]  ⍝
[6]  ⍝ ENSURE <A> IS A MATRIX ∈ SETUP RESULT
[7]   R←((B←1⍴⍴A←(¯2↑ 1 1 ,⍴A)⍴A),3)⍴0
[8]  ⍝ PASS CONTROL TO SEQUENCE VECTOR
[9]   →(B←(B⍴A),B)[C←⎕IO]
[10] ⍝ EXECUTE EACH ALGORITHM <RELTIME> TIMES
[11] A:R[C;]←RELTIME CPUTIME A[C;] ◇ →B[C←C+1]
[12] ⍝ DONE; MAKE 1ST COL. INTO INDEXED VALUES
[13] B:R[;⎕IO]←R[;⎕IO]÷⌊/R[;⎕IO]
[14] ⍝ CREATE FORMAT STRING FOR ⎕FMT
[15]  C←(⍕1↓⍴A),'A1,X2,F5.2,X2,F7.2,X2,F8.5'
[16] ⍝ CREATE DISPLAY TABLE
[17]  R←C ⎕FMT(A[B;];R[B←⍋R[;⎕IO];])
[18] ⍝ PUT ON COLUMN HEADINGS
[19]  R←((-1↓⍴R)↑'INDEX      MEAN    STD DEV'),[⎕IO] R
    ∇
```

The columns generated for the mean and standard deviation are done so as to place the indices in the proper context. Many features in the aforementioned functions are specific to SHARP APL and would have to be modified to run on different APL systems.

While there are many general principles to follow in order to design a system that will run efficiently, it is sometimes necessary to make specific measurements for a specific system. The procedures given in this section make it possible to do accurate measurements when they are needed.

CONCLUSION

Efficiency is more than just exploiting quirks and loopholes to squeeze out CPU's from an application. When considered in the design process, it can eliminate the need to optimize or redesign at a later time.

An efficient design is one which considers how the application is used. For example, as discussed in the section on structures, a data base which is efficient for data retrieval may not be effective for data searching. This is a problem if it is typically used that way. In reviewing the many methods of "flow control" we found that some are faster than others. Their relative efficiency depends on the context in which they were used.

The broad topic of efficient design is one which cannot be *fully* covered by any document. Efficiency is a fluid notion, both because of changing APL systems and because of changing ideas. Thus, it is difficult to pronounce a system as completely efficient; though it may be as efficient as currently possible.

Notes

1. In an attempt to simplify the process of quantifying the work APL performs, several liberties have been taken with respect to the segments of cost involved. In the sample data base records are stored in matrix form with rows as records and columns as fields. The data itself is a non-descript *item* which allows us to remove all aspects of type and rank from the discussion. Furthermore, idiosyncrasies of the respective file systems are ignored such as *growing replace* costs and the benefits of *cache memory* operations.

   For the purpose of this discussion, a **tight** APL loop is one in which a minimal amount of work is being performed. This means the cost of catenating records after retrieval is *not* being considered. Because of the many methods by which records can be stored after extraction, whether by catenation or by indexing or even by appending to another file, this aspect has been conveniently overlooked.

2. Typically, data is stored on a device blocked to a specified maximum number of bytes per physical block. The SHARP APL file system uses a physical record size of 3156 bytes and data is stored across the ⌈*ACTUALBYTES*÷3156 number of blocks. Therefore, to keep things simple, the coefficients generated by the formulae in this section are based on a physical record size of 3156 items.

3. Cheung, To-Yat. "Estimating Block Accesses and Number of Records in File Management". *Communications of the ACM 25, 7* (July 1982) pp. 484-487.

   Yao, S.B. "Approximating Block Accesses in Data Base Organizations". *Communications of the ACM 20, 4* (April 1977) pp. 260-261.

   The following function generates the number of blocks accessed given a data base of *REC* records partitioned into blocks of *REC*Δ*BLK* records based on the work of Yao as above.

```
     ∇ R←REC△BLK YAO REC;A
[1]  A←(1+REC)-⍳REC
[2]  R←A∘.-,REC△BLK
[3]  R←×⍀R÷⍉(⍴⍴R)⍴A
[4]  R←1-R[⍳REC;]
[5]  R←R×(⍴R)⍴REC÷REC△BLK
     ∇
```

4. For a full description of the SHARP APL implementation of event trapping, see Chapter 13 of the *SHARP APL Reference Manual* by Paul Berry.

5. The following function when used with a file of packaged data, should sufficiently handle the *housekeeping* involved with paging and executing objects *within* the environment of the package:

NOTES

```
    ∇ MASTER;MASTER;A;B;C
[1]  ⍝ TIE FILE (IF NOT TIED) TO SIGNON TIME
[2]   →(⎕NUMS=C←⌶24)/A ◇ 'PAGEFILE' ⎕STIE C
[3]  ⍝ GET OPTION FROM USER, EXIT IF NOTHING ENTERED
[4]   A:⎕ARBOUT 0⍴⍞←'OPTION: ' ◇ →(⍴B←(' '≠B)/B←⍞)↓0
[5]  ⍝ FIND ALLEGED BUCKET
[6]   A←1+(--/2⍴⎕SIZE C)|+/(⎕AV⍳B)-⎕IO
[7]  ⍝ IS IT REALLY THE RIGHT PLACE?
[8]   →(B≡'WSID' ⎕PVAL A←⎕READ C,A)↓A
[9]  ⍝ BUILD THE HEADER FOR A LOCAL FN
[10]  B←'MASTER ⎕ER;MASTER;A;B;C;A;B',,';',⎕PNAMES A
[11] ⍝ FIX FULLY CONSTRUCTED LOCAL FN
[12]  →(⍴⍴B←⎕FX B,[⎕IO-0.5](⍴B)↑'⎕PDEF ⎕ER ◇ ⍎⎕LX')↓A
[13] ⍝ IF ⎕FX WORKED, RUN THE 'PACKAGE'
[14]  MASTER A ⎕PINS '⎕ER' ⎕PACK '' ◇ →A
[15] ⍝ ⎕FX DIDN'T WORK OR OPTION DOESN'T EXIST
[16]  B: 14 ¯10[⎕IO+0=1↑0⍴B]↓'UNABLE TO RUN OPTION NOT FOUND'
[17]  →A ⍝ GO BACK AND ASK AGAIN
    ∇
```

For a description of packages and related system functions in SHARP APL, see Chapter 29 of the *SHARP APL Reference Manual* by Paul Berry. The function ≡ above returns a scalar one or zero if the two arguments are identical or not.

References

Abrams, Philip S. "Program writing, rewriting and style." *Proceedings, APL Congress 1973.*

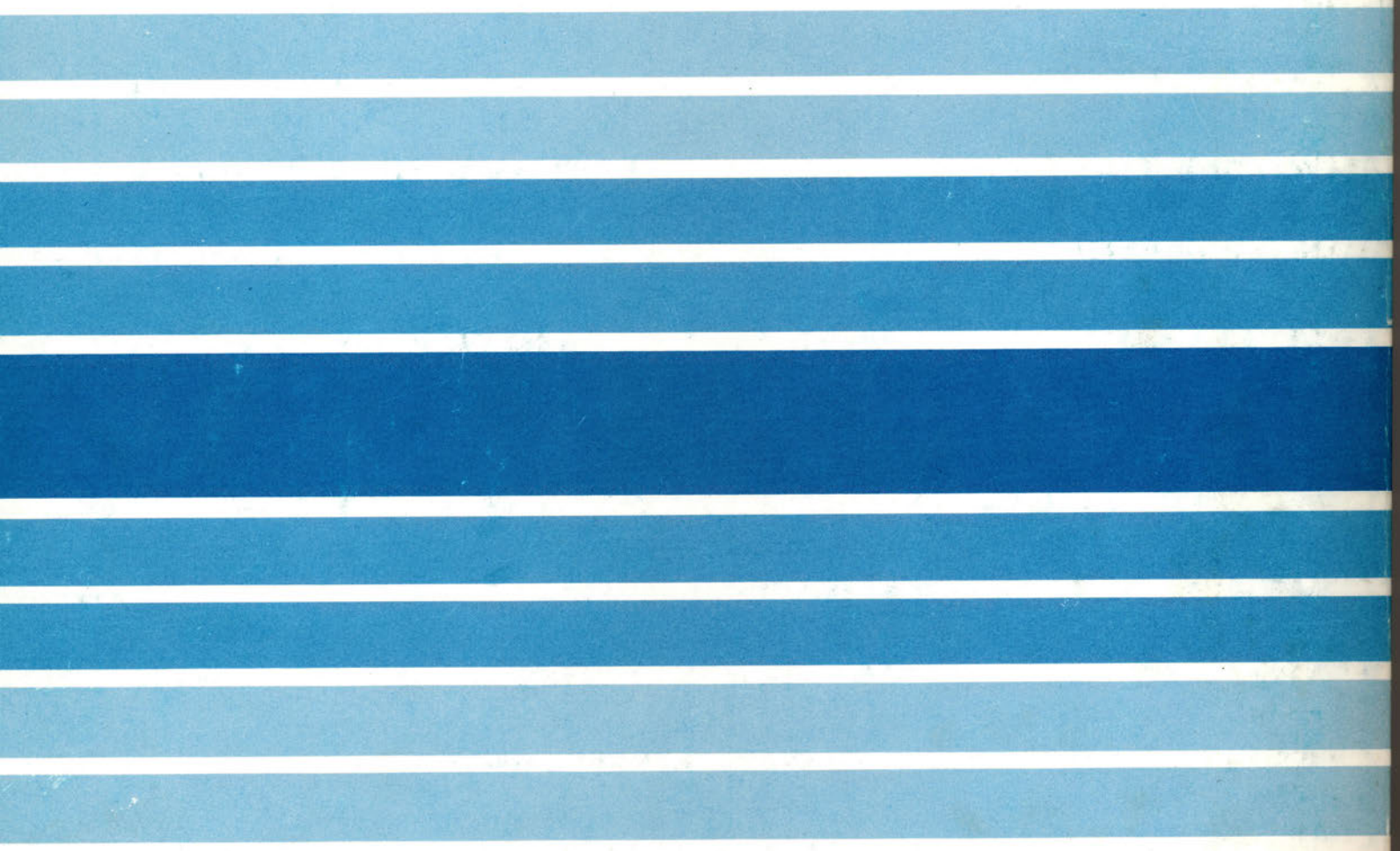
Iverson, K.E. *A Programming Language.* John Wiley & Sons, 1962.

Kent, C. "Function Timing." *I.P. Sharp Newsletter.* August 1975.

Knuth, Donald E. *The Art of Computer Programming.* Vol. 1: Fundamental Algorithms, Chapter 1, and Vol 3: Sorting and Searching, Chapters 4 & 6. Addison Wesley Publishing Co., 1973.

Roden, W.J. Jr., and G.H. Foster. "Execute And Its Use". *APL Quote Quad, Vol. 12, No. 1, ACM APL81 Conference Proceedings.* September 1981, pp. 263-269.

Yourdon, Edward, and Larry L. Constantine. *Structured Design*, Chapter 15.



Printed in Canada